Table of Contents

Contents:

Panther

Application Development Guide

Prolifics

Release 5.51

Document 0404
March 2017

Copyright

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	

TABLE OF CONTENTS

Contents:

About This Document

Application Development Guide describes an application development path, starting with project setup and configuration through to packaging and deployment. Covering various topics related to application development, it discusses approaches to development, strategies for using Panther effectively, and the order in which tasks should be performed.

This guide is organized in the following sections:

Overview: Building a Panther Application
: A comprehensive overview of Panther and its application development process.

Section One: Preparing for Development
: Topics include the organization of your Panther distribution, a discussion of project requirements, how to set up your application servers and development clients, how to initialize and connect to your database engine and middleware, how to access your application libraries.

Section Two: Creating Application Building Blocks
: An introduction to Panther's application components, including screens, widgets, repositories, menu bars, reports and service components.

Section Three: Preparing the Programming Interface
: Information about programming events in Panther and how to use JPL, C and Java for event processing, accessing widgets programmatically and manipulating the screen sequence in your Panther application.

Section Four: Accessing the Database
: The protocol for Panther's interaction with your database engine-how data and status information is fetched from, or written to, the database, how to build screens that use the transaction manager, how the transaction manager gets its information and processes transactions, and how to customize your transaction manager applications.

Section Five: Testing Your Application
: Description of Panther's built-in debugger and instructions for using it to debug your application.

Section Six: Deploying the Application
: Information on building Panther development and production executables and for packaging your Panther application for distribution.

Section Seven: Advanced Development Topics
: Topics related to Panther's hook functions, portability and internationalization.

Section Nine: Appendices
: Information about development utilities. Also includes descriptions of Panther's sample applications.

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.

Convention	Item

`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.
`[ ]`	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`*`]...`

Convention	Item
. . .	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

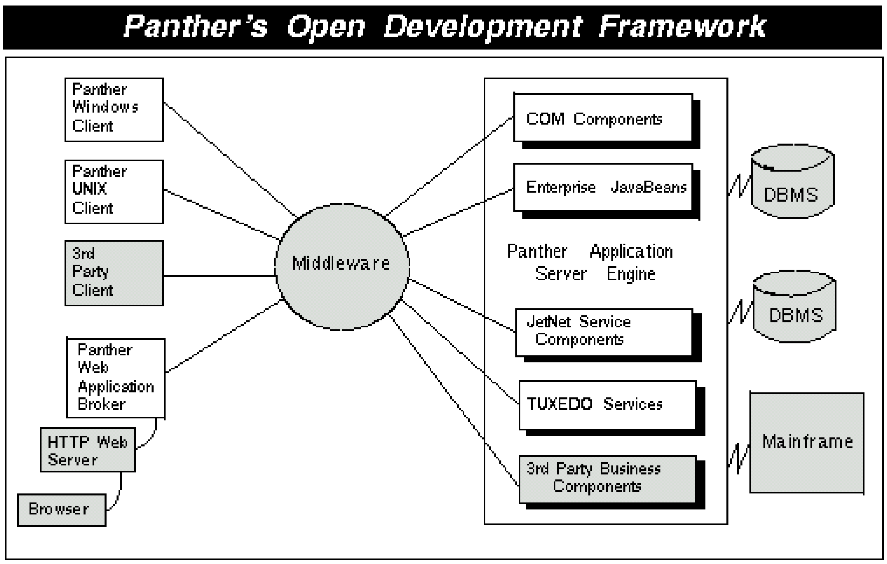
When contacting Customer Support, be prepared to provide the following information:

- 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 Building a Panther Application

Panther provides a framework for application development—for a simple, single client and server operating on one machine, or for distributed applications that are more complex and demanding in terms of architecture and performance requirements. Panther is designed to suit your immediate needs and can grow with the enterprise's requirements.



This chapter provides a summary of the steps for developing a Panther application. Although the information in this overview describes a particular stream of development, the Panther framework is flexible and makes no demands on exactly how, or in what order, the development process is approached. Its open style of development allows you to use the methods needed by your application and your development team.

Application Development Steps

Installing Panther

The *Installation Guide* provides instructions for installing the following Panther products:

- Two- or three-tier Panther clients, either UNIX or Windows.
- Panther web application server.
- Panther application server for your chosen middleware.
- Panther database drivers.

Designing the Project Requirements

Define the Application Architecture

With Panther you can start with a simple, client/server, two-tier architecture application and when your application requirements increase, convert your application to n-tier architecture. Or, begin with a distributed application structure—create both the client and server components at the same time.

A two-tier application is distributed between client machine and the database server. The client machine controls the presentation logic and all or most of the application logic. The client can access data in the database through SQL interactions (coded or application-generated) or by making calls to the database's stored procedures.

Distributed applications separate the business logic of the application from the user interface. The application logic is built into service components that reside on the application or component server. The client workstations provide the presentation interface, some application logic, and can make requests to the application servers to perform commonly used logic and access a database. Using service requests, there is no need for a continuous connection between the client and the database server.

Define the Database Access

Before starting development, you need to design your database schema and decide which type of database access you will use in your Panther application. Panther's database interface allows you to write your own SQL statements or call the database's stored procedures. You can also use the transaction manager to automatically generate the SQL statements needed for a screen.

Define the Client Platforms

You need to decide which client environment is needed by the application. Panther applications support a wide variety of interfaces including Windows, Motif, Unix character-terminals and web browsers.

Define the Development Team

Panther facilitates team development and enables a development organization to leverage the individual skills of their team members:

- Design Team—The designers are responsible for modeling the business processes that will be reflected in the application and for defining the project requirements. For component-based development, the business processes are then translated into application components.
- Server Developers—The application developers responsible for the server-side of the application focus on working with the database engine using SQL and stored procedures, the application coordination with the middleware, and the business logic of the application. Panther provides application building blocks, in the form of class libraries, that will jump-start development and enable application developers to focus on the business logic of the application rather than writing the low level code for the underlying architectural technologies.

- Client Developers—The application developers responsible for the client-side of the application focus on presentation and user interfaces.
- Web HTML and Graphics Page Design—When creating web applications, the presentation layer can be created by a distinct team that uses any industry-popular web page publisher (for example, Microsoft FrontPage) to create the HTML and graphics for the visual browser display. Because Panther allows you to encapsulate Panther objects into pre-built HTML pages, the application development team can be separate from the HTML and graphics team.
- Quality Assurance—QA is involved when entering the testing and debugging phase of a project cycle.

Configuring the Server Environment

In three-tier applications, the application server contains the application logic segmented into service components and makes those services available to the application clients as well as connecting to the data resources the application needs.

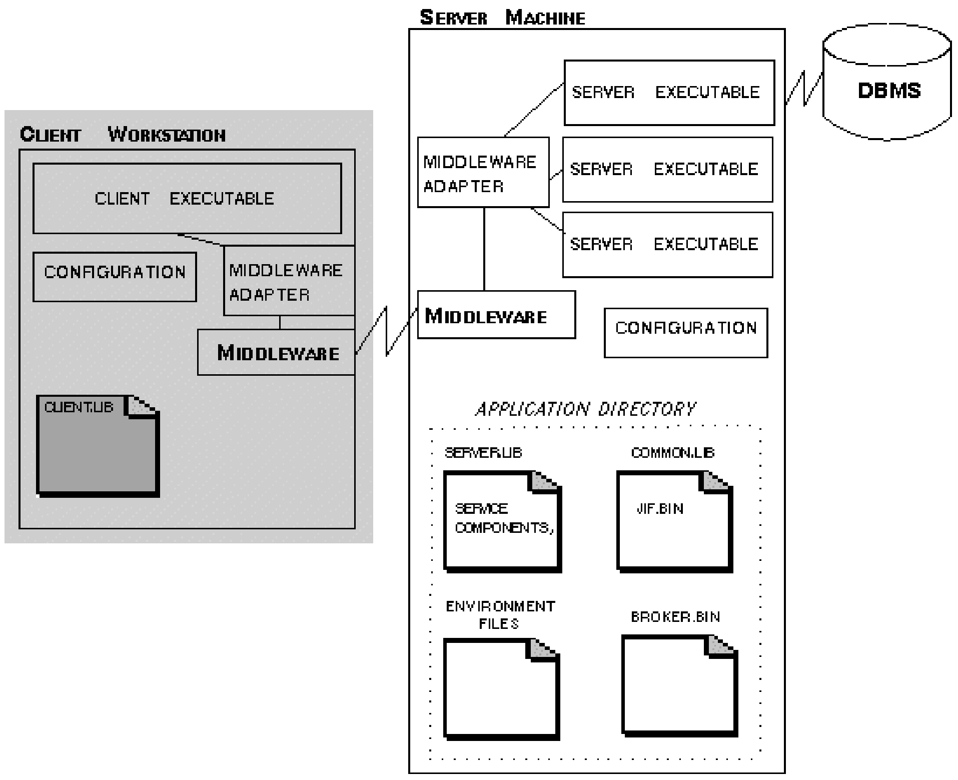


Figure 1-1 Components of an application server in a Panther JetNet application.

You will need to create an application directory for each application on the application server machine. In that directory you can find:

- Application libraries, such as `server.lib`.
- Environment files used to specify the location of the Panther installation with its Panther software tools, the license file, and any other settings needed by the application server.
- For the JetNet and Oracle Tuxedo applications, a middleware configuration file. By default, this file is named `broker.bin`.

For JetNet and Oracle Tuxedo applications, you must also configure the types of servers: standard servers for three-tier development and production environments, a conversion server to run any three-tier applications you converted from a two-tier architecture, and file access servers used for both two- and three-tier development to provide the development team access to remote libraries and repositories.

Refer to Chapter 5, “Preparing the Application Server,” for additional information on preparing the application directory and the server environment.

Setup the COM Component Server

For COM/MTS applications, set up the machine for deploying your COM components during development. Create the application directory containing server.lib and specify that directory in the editor so that the DLLs, type library files and client registration files will be saved to the proper location.

Start with Chapter 1, “Overview,” in *COM/MTS Guide* for information about COM settings and deploying COM components.

Preparing the Development Environment

You need to prepare the development environment for your chosen application architecture.

Connect to the Database

In order to access a database, the database engine must be initialized in the Panther executable, and the application must declare a connection to the database. In two-tier applications, the application client makes a direct connection to the database. In three-tier applications, the Panther application server maintains the database connection.

For more information, refer to Chapter 7, “Initializing the Database,” and Chapter 8, “Connecting to Databases.”

Setup the Application Client

The application client in both two-tier and distributed applications contains the presentation portion of the application. After installing the Panther client pertaining to your application architecture, you need to specify the location of Panther setup files and application libraries. You can also set variables that control the behavior of part of a Panther application, such as how the cursor behaves.

Configure Library Access

Developers work using local client libraries located on the client workstation or host machine. In JetNet and Oracle Tuxedo applications, developers also have the option of using remote client libraries located on the application server.

Refer to Chapter 10, “Accessing Libraries,” for information about libraries and using source code control programs.

Connecting to the Middleware

For three-tier applications, the middleware allows the application's clients to interact with the application's servers. The clients and servers can reside on different machines, connected by a network; the middleware API interfaces with the middleware, allowing interaction between machines.

The main tasks performed by the middleware are:

- Establish client connections to the application. For the JetNet and Oracle Tuxedo middleware adapters, use the `client_init` command.
- Forward service requests to the middleware, where they are forwarded to an appropriate application server for processing.
- Return data and status messages to a client following a service request.

For more information, refer to Chapter 9, “Connecting to the Middleware.”

Defining Services

For three-tier and COM applications, you need to build service components that implement the interface and business logic of your application.

Services in JetNet/Oracle Tuxedo

In JetNet and Oracle Tuxedo applications, services are subroutines that do the work required for an application to access a resource manager, usually a database. They are invoked by service requests made by clients or other services.

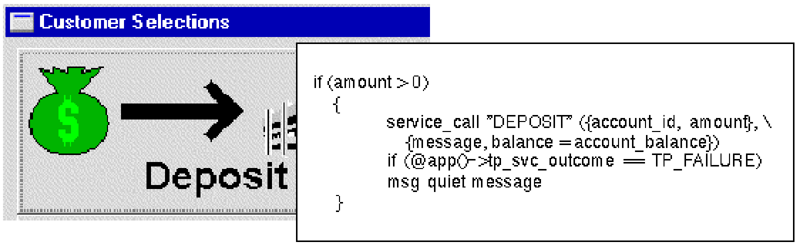
Service routines are responsible for receiving data from the client (if sent), performing some task, and returning data to the client (if requested).

To expedite responses to service requests, the application can run multiple instantiations of a server. Service requests are routed to servers in the way that provides the fastest response.

A service can consist of three parts:

- A routine that implements the service.
- A service component (optional) that provides a physical means of sending, receiving, and processing data.
- A service definition in the JIF.

To send and receive data between a client and application server, use `service_call` and `service_return` in conjunction with the `receive` command. The `service_call` command initiates a service request, as illustrated in this excerpt from a deposit process:



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

For JetNet and Oracle Tuxedo applications, the JIF is the central storage file of information about your application's services and queues. Panther accesses it at runtime to determine the requirements and specification of services and queues for three-tier processing.

A service is defined in the JIF by its name, the type of transport buffers it uses—that is, in what form information is passed to and from clients—and the name of the service component that uses the service.

You can also create service groups which can facilitate how services are made available on servers. For the Oracle Tuxedo middleware adapter, you can define reliable queues. Reliable queue definitions include their transport buffers and the names of associated reply and failure queues, if appropriate.

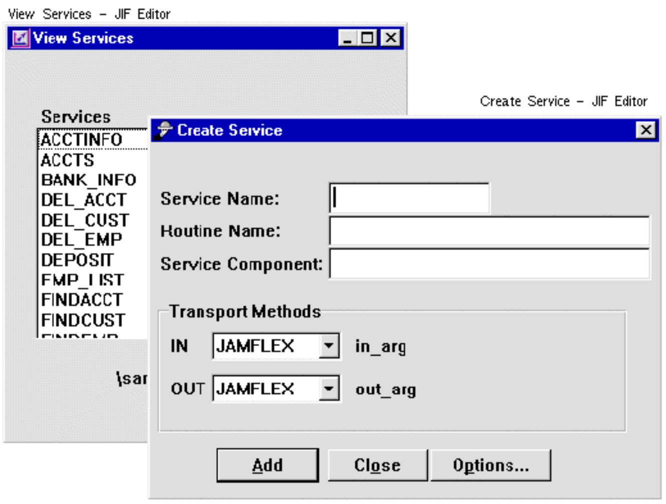


Figure 1-2 The JIF editor defines the service: its service name, corresponding routine name, service component, and transport methods.

Once you determine what services are required for your application, use the JIF editor to create the JIF for your application. For instructions on how to use the JIF editor, refer to Chapter 24, “JIF Editor,” in *Using the Editors*.

Services in COM and EJB Components

For COM components and Enterprise JavaBeans (EJBs), you create service components in the editor and save them in a server library. Part of that editor process is defining the component interface and specifying the properties and methods implemented by the component. Once the service component is available, client screens can use JPL, C, or Java to instantiate the service component and access its methods and properties.

Since you specify the type of component system before instantiating the component, you can use the same programming interface for COM and EJB components.

A client calls the component's methods using `sm_obj_call` and accesses its properties using `sm_obj_get_property` and `sm_obj_set_property`. The component uses the `receive_args` and `return_args` commands to receive the client's data and pass data back to the client.

For more information about building and deploying COM components in a Panther application, refer to Chapter 3, “Building COM Components,” in *COM/MTS Guide*.

For more information about building and deploying Enterprise JavaBeans in a Panther/WebSphere environment, start with Chapter 5, “Building Enterprise JavaBeans,” in *Panther for IBM WebSphere Developer's Studio*.

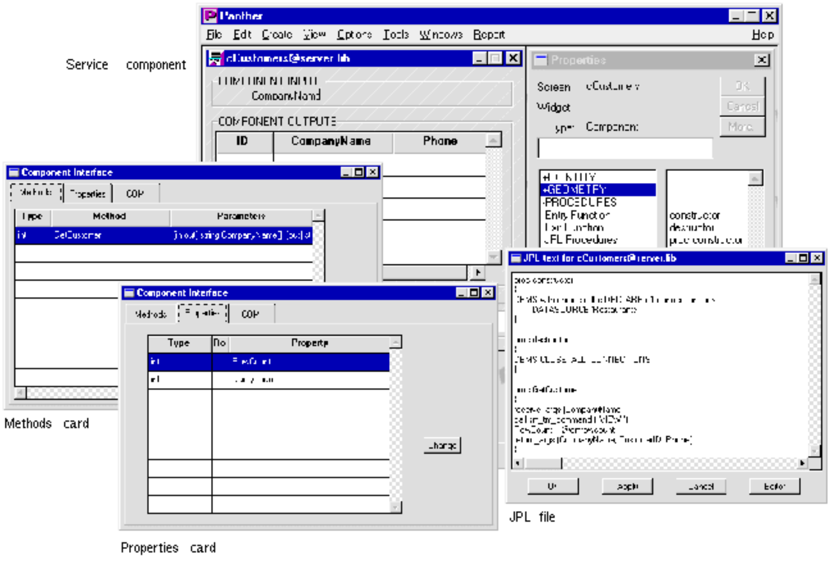


Figure 1-3 In the Panther editor, you can define a service component's methods, properties, and JPL programming.

Building a Repository from a Database

Once you define your database contents, that is, the tables, columns, and primary and foreign keys, and you are connected to the database from within the editor, you can import the definitions into Panther.

The import process populates the open repository with one repository entry for each imported table. Figure 1-4 illustrates three repository entries that are the result of a database import.

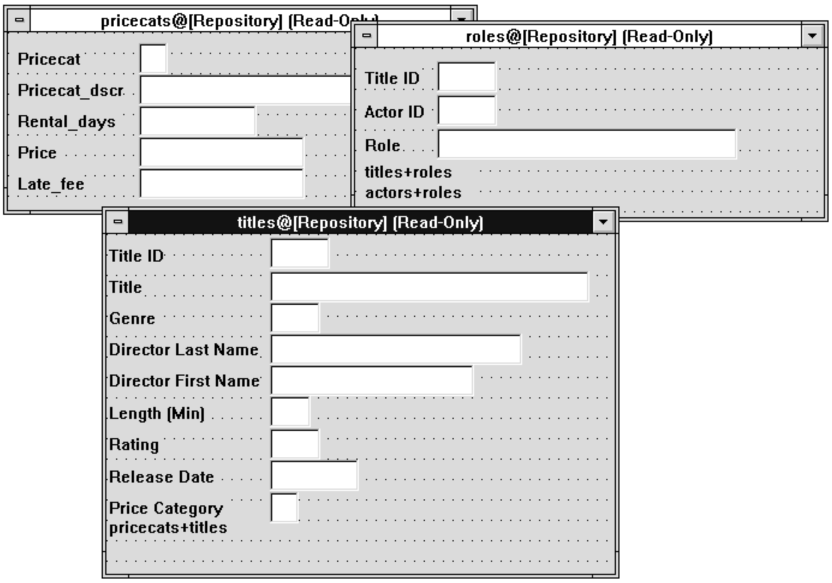


Figure 1-4 Repository entries created from database tables.

The repository entry that results from an import contains one text widget and one label widget for each database column in a table. Properties are automatically set to reflect the properties of the column in the database. You can reimport data definitions whenever you need to, and thereby update the information stored in the repository. In turn, any application objects you built using repository objects are also updated—changes to database column properties can be propagated throughout your application via inheritance relationships.

Refer to “Creating and Opening a Repository” on page E-19 in *Using the Editors* for details on creating a repository.

Refer to Chapter 11, “Creating and Using a Repository,” for information on uses of the repository.

Table Views and Links

Database table information is stored in a table view widget. This provides Panther with the information it needs to access the database and understand how database tables are related. Table view properties include the name of the database table, the table's columns, and the columns that compose the table's primary key. The import process creates a table view for each imported table, and adds the widgets corresponding to columns to each table view.

Table relationship information is stored in link widgets. The import process creates links based upon foreign key information contained in the database. If the database contains no foreign key information, then you can create the links manually in the editor.

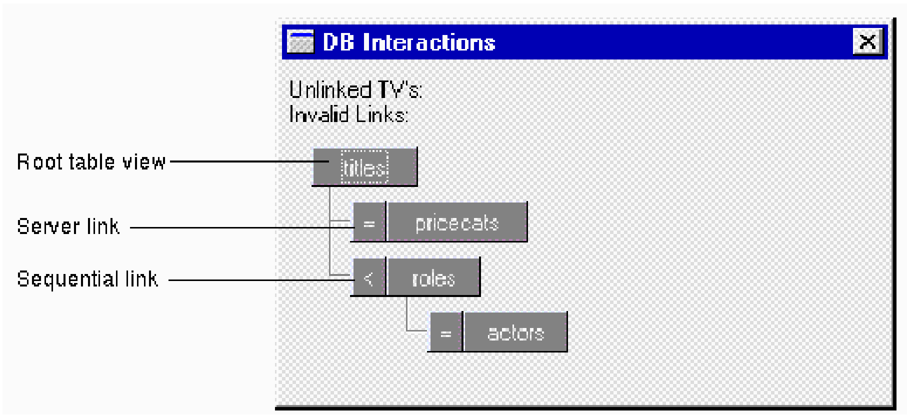


Figure 1-5 The DB Interactions window displays the table views used on the screen and their relationships.

Creating Application Components

Use the editor to gain access to all of Panther's authoring tools. Either with the wizards or from scratch with the editor, build the client screens that define the application's user interface, the service components, and the reports needed by your application.

A Panther application consists of libraries to which you should have direct access when you start up the editor:

- The client library (`client.lib`) for storing the objects that make up the user interface, such as screens, menu bars and tool bars, JPL procedures used by the client, a styles file (`styles.sty`) and images that illustrate push buttons and toolbar items.
- The server library (`server.lib`), required only for distributed and three-tier applications, for storing service components and service-specific JPL procedures.
- (JetNet/Oracle Tuxedo only) The common library (common.lib) for storing objects that are needed application-wide, such as the JIF and JPL that is used by both client and server.

Libraries can be named anything you wish and configured to be available during development and/or deployment of your application.

Repository Development

The visual object repository is used during development to define and store a set of objects needed to build screens, service components, and reports. Once the repository is populated, you can easily make a new application component by copying the necessary objects from the repository.

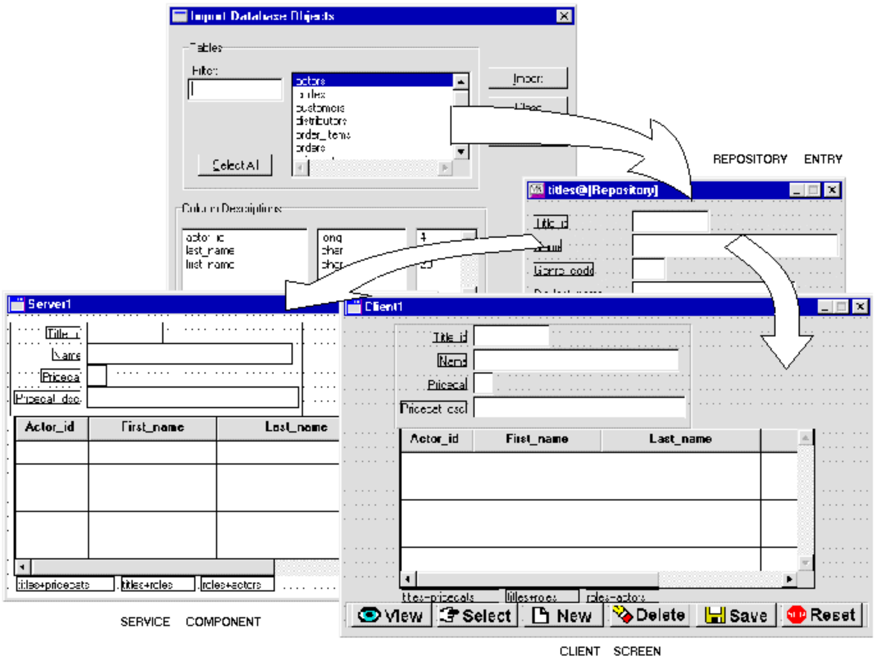


Figure 1-6 Inheritance from database to repository to screens.

In addition to the development time saved by creating objects only once, the repository can be used to easily update application objects by using inheritance. When you copy an object from the repository, the copy, or child, retains the property definitions of the original object, the parent. If you change the properties of the parent object in the repository, the properties that the child has inherited are also updated.

You can enhance repository members without breaking the ability to reimport database tables. Your enhancements might include:

- Changing the widget size.
- Changing the labels.
- Adding new widgets.

For example, you could include a total price widget to hold the product of the unit price and the quantity database columns; the code associated with the widget to make the calculation is stored with the widget.

You can also enhance the repository by storing commonly used application objects, such as OK and Cancel push buttons. These frequently used objects can be made available to the development team through shared libraries.

Graphical Editor

Invoke the editor to create screens, reports and service components. To facilitate the process of building database-related screens, you can use the screen wizard. In addition to building the user interface, you can build service components to handle the processing of service requests in distributed applications. You store the client and server components in their appropriate Panther libraries.

Typically, the editor starts with a single, empty, screen. The screen can be enhanced by adding widgets from tools you select from the tool bar or from the menu bar. Since you can have many screens open at once, you can easily copy objects between screens via drag and drop.

Screen Wizard

The easiest way to create new screens is by using the screen wizard. This option is available each time you choose to create a new screen in the editor. The screen wizard takes full advantage of the contents of your repository when you use it to build screens and, for JetNet/Oracle Tuxedo applications, service components.

The screen wizard can create three basic types of screens: master, master-detail, and master-detail-subdetail. In each section of the screen, you can have one or more table views using either a single row or grid layout. When the screen is completed, it contains the widgets and their labels, a series of push buttons or a menu bar for commands, selection screens to facilitate data entry at runtime, and a JPL module containing the procedures needed to execute basic database transaction commands.

For JetNet/Oracle Tuxedo applications, you can choose to create both a client screen and service component using the screen wizard. They look very similar, but the client screen resides on the client, describes the user interface, and interfaces with the database by way of service requests, while the service component resides on the server.

Enhancing the Interface

Once the basic screen design is complete, you can:

- Modify properties to set default appearance and behavior of widgets; for example, implement double-click events in a list box, alter the automatic SQL generation by changing database properties, add colors to widgets or decorations to the screen, such as boxes or lines.
- Add additional widgets: push buttons to define user actions or radio buttons and check boxes to present a group of available options.
- Include business graphs or ActiveX controls as another means of displaying data.
- Create menus and toolbars using the menu bar editor.

For a description of available widget types, refer to "Types of Widgets" on page 2-19 in *Using the Editors*.

Modify Properties

Widget, screen, report and service component properties can be modified during development in the editor or programmatically at runtime. Even though most properties are available through the Properties window, runtime-only properties must be changed programmatically. At runtime, there are also application properties which set the default behavior for the entire application.

For an alphabetical list of runtime properties, refer to Chapter 1, "Runtime Properties," in *Quick Reference*.

Define User Actions

Screens created by the screen wizard already include push buttons and/or menu bars to fetch and update data using the transaction manager, the Panther component that handles database events. Additional widgets can be created, or the functionality of existing widgets can be modified as necessary according to your application needs.

Define Event Functions

In addition to defining user actions, you can specify processing to occur on application events. One way of looking at a Panther application is as a series of application events. When a screen opens, a sequence of screen events occur. Then the first widget on the screen gets focus; a sequence of widget events occurs. Only then is the screen ready for user input, which in turn launches another sequence of application events.

Each of these application events can have JPL, C, or Java processing associated with it, to be invoked when the event occurs. The processing can be for a specific object, such as a single screen, or application-wide for all objects. When processing is application-wide, the function must be written in C and installed in `funclist.c`.

When processing is specified for a specific object in JPL or C, then the name of the C or JPL function is entered as a property of the object. For example, to invoke myfunc whenever a widget gains focus, specify myfunc as the value of the Entry Function property of the widget.

For Java, the Java Tag property allows you to associate the name of a Java class to a Panther object. The specified Java class is an event handler specifying methods for the events supported by the object.

For the JetNet middleware adapter, there are additional events and event handlers. Event handlers are called when middleware API events take place in your application, for example, when a client makes a service request or when a service is advertised. Handlers can invoke other services as well as open, select, or close service components as needed. Default event handlers are provided; you can also write and install your own event handlers.

For more information:

- Chapter 17, “Understanding Application Events.”
- Chapter 20, “Writing C Functions.”
- Chapter 21, “Java Event Handlers and Objects.”
- Chapter 6, “JetNet/Oracle Tuxedo Event Processing,” in *JetNet/Oracle Tuxedo Guide*.

Fetch Database Information

Database operations in Panther applications are processed by the following software components and provide you with different levels of database access:

- Transaction manager—Determines what SQL must be generated and executed, and asks the next level to do the work.
- SQL generator—Constructs SQL statements and asks the next level to execute them.

- Database interface—Passes SQL requests to the database, and returns formatted results to Panther. The database interface is implemented via the `dbms` verb in JPL and the C library function `dm_dbms`.
- Database API—Provided by the database vendor.

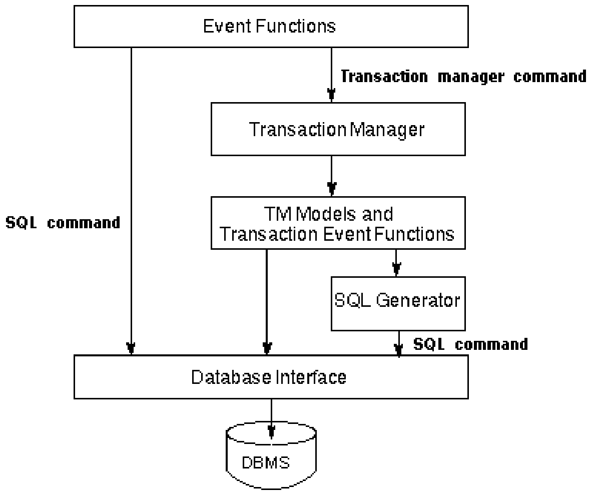


Figure 1-7 The relationship between your application, Panther components, and your database.

Screens built with the screen wizard use the transaction manager to fetch and update database information. From the widget properties, events in the transaction manager can generate the SQL needed to populate the screen. When the data is displayed on the client, the transaction manager keeps track of any changes made to the data and generates the SQL needed to update the database.

For applications not using the transaction manager, you can write your own SQL statements or use the database interface to call stored procedures on the database engine.

For more information about database operations, start with Chapter 27, “Performing Database Operations.”

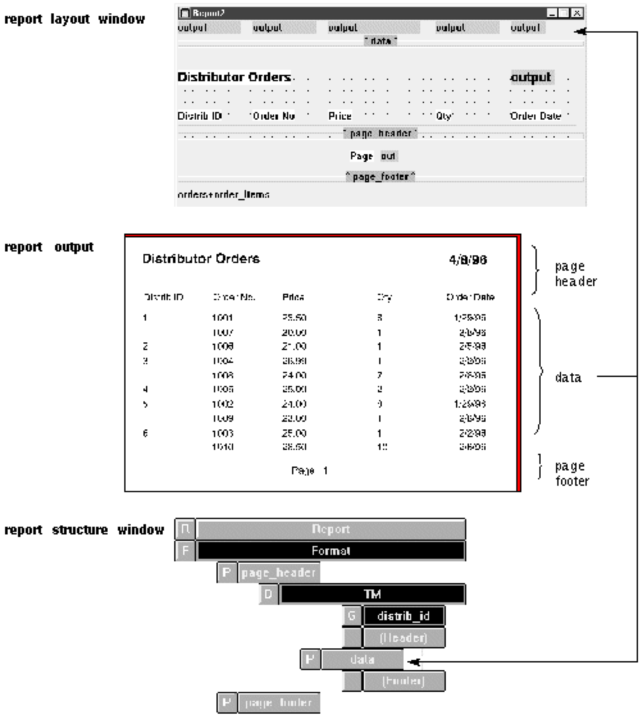
Add Reports

With Panther, you can build reports to supplement your application processing. Once created, a report can be invoked from an application, from a web browser, or on the command line, and can be output to the screen, to a printer or to a file.

A report definition has two windows in the editor: the report layout window and the report structure window. The report layout window, containing one or more layout areas, defines the report content. Each layout area contains widgets whose properties define the source of report data. A widget's position in the layout area determines its position in the report output.

The report structure window, consisting of a series of nodes, determines the order of report processing. Each layout area must have a corresponding print node in the report structure in order to appear in report output. Other nodes define the format of the report, the properties of report groups, and the programming actions to take during report processing.

For information on using the report wizard, refer to Chapter 5, “Report Wizard,” in *Using the Editors*. To build reports in Panther, start with Chapter 1, “Overview of Panther Reports,” in *Reports*.



```
Distributor Orders                                          4/8/98

Distrib ID    Order No.    Price               Qty        Order Date
1             1001         25.50               3             1/25/98
              1007         20.00               1              2/6/98
2             1006         21.00               1              2/5/98
3             1004         26.99               1              2/3/98
              1008         24.00               7              2/6/98
4             1005         25.00               2              2/3/98
5             1002         24.00               9             1/26/98
              1009         22.00               1              2/6/98
6             1003         25.00               1              2/2/98
              1010         28.50               10             2/6/98

                          Page  1
```

Figure 1-8   The report layout window and the report structure window work together to define a report.

Integrating Application Components

To integrate your application components, you can:

- Define screen interaction by connecting your screens to one another, invoking functions, and initiating communications between clients and servers.
- Use a variety of methods to share data between screens.
- Attach event functions or event handlers to control specific events.

Define Screen Interaction

Menu items, push buttons, and function keys can be associated with processing logic. In all three cases, this association can be defined with Panther control strings. Push buttons and menu items have a Control String property. You can also define control strings for the screen or application by associating a control string with a function key or Panther logical key.

When a user of your application chooses a menu item or clicks on a push button, the control string attached to the object can:

- Execute a function or JPL procedure.
- Close the current screen and open another.
- Open another screen on top of the current one.
- Invoke a system command or program.

For detailed information about using control strings, refer to Chapter 18, “Programming Control Strings.” For details on screen management, refer to Chapter 24, “Setting the Screen Sequence.”

Share Data Between Screens

Panther applications typically require data to be shared between screens. There are several methods you can use.

Sending and Receiving

This technique, used only for client screens and implemented by the JPL statements `send` and `receive`, is the most modular of the data sharing techniques. The sender specifies exactly what data is sent. The receiver specifies exactly what data is received. The order of the data items determines the correspondence between items sent and received. There are also Panther library functions you can use for sending and receiving.

Explicit Reference

In JPL you can refer explicitly to a data item on an open window by using the notation `@scr(screenName)->widgetName`. For example, you could copy data from `empid` on the `empsrch` screen to empid on the active screen by writing:

```
empid = @scr(empsrch)->empid
```

Local Data Block

The local data block (LDB) enables sharing of data automatically based on widget names. Each time a screen becomes active, its widgets are populated with data from those LDB fields that have corresponding names. When the screen is made inactive, Panther copies widget data back into the LDB. This eliminates the need to write code to move data between screens.

You can create multiple LDBs. An LDB exists until it is explicitly destroyed. This allows access to data regardless of whether or not it is on the active screen, or on any open screen.

The LDB contains data that is local to each user, but that is globally shared within a user's invocation of an application. Use the LDB to store data that is shared between many screens when you want changes on any one screen to cause changes on all screens that share the same widget name.

JPL and C Variables

You can move data between screens by saving the data in JPL or C variables, opening the destination screen, and copying the data into the destination screen.

For details on moving data between screens, refer to Chapter 25, “Moving Data Between Screens.”

Setting Application Security

In three-tier applications, the application server connects to the database resources, allowing access to the data needed for your application. To make the application secure, you need to control the client's access to the application.

In two-tier applications, the database engine specifies the users and passwords that must be used to access the database.

Deploying on the Web

In web applications, screens are requested via a URL in a web browser. Then, if the screen is submitted back to the web application server, the screen is processed, any service requests or SQL needed to fetch information are generated, and then the data is sent back to the browser requesting the information. In n-tier applications, the web application server operates as a Panther client.

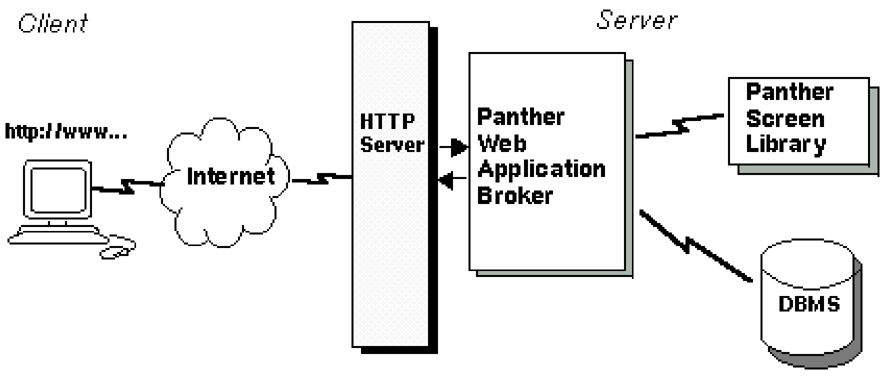


Figure 1-9 When the HTTP server receives a request from a browser for a Panther screen, the request is sent to Panther for processing.

Configure the Web Application

The Web Setup Manager creates an application directory on the web application server and creates the initialization file needed for each web application. This initialization file contains the settings used by the web application.

For information on running the Web Setup Manager, refer to Appendix B, "Web Setup Manager," in *Web Development Guide*; on setting up a Panther web application, Chapter 2, "Web Application Setup," in *Web Development Guide*.

Create the Web Client Screens

You can view any of your application screens on the web, but you might want to build separate screens which take advantage of web-specific features and cache data in order to operate in the web's stateless environment.

For more information on building web applications, including how screen and widget properties operate differently in the web environment, refer to Chapter 3, "Setting Properties for Web Applications," in *Web Development Guide*.

Program Web Events

Application logic for web-specific events can be written in JavaScript or VBScript and performed in the browser at runtime. In addition, JPL, C and Java programming can be run on the web application server. For more information, refer to Chapter 9, “Using JavaScript and VBScript,” in *Web Development Guide*.

Testing and Debugging

User Interface

In the editor, you can test the behavior and appearance of client screens built for both two- and three-tier architecture. For two-tier architecture, you can connect directly to the database, and see how your screen interacts with "live" data. For three-tier architecture, you need a connection to the database by way of the middleware. The service component, if there is one, must also be saved to the server library, so that three-tier access is possible.

You can also use Panther's debugger to examine the internal behavior of the client screens—for example, walk through JPL procedures and watch values being set in variables and fields on the screen.

For instructions on using the debugger, refer to Chapter 39, “Using the Debugger.”

Service Components

For three-tier applications, the service components reside on an application server. There is, in practice, no visible way from the user's perspective to see what is happening on the server. However, you can test and debug a service component's behavior in much the same way you test a client's. Connect directly to a database, and go into test mode from the editor.

To test whether the service component and its JPL routines function with its corresponding client screen, you must save the service component to the server library and test the client screen that calls the service provided on the service component.

For JetNet/Oracle Tuxedo applications, if you have a debuggable server running, you can view the service component or walk through JPL service routines.

Fine-Tuning the Application

Using C and Java Code

Panther provides an interface to functions written in C and Java so that you can:

- Integrate third-party software with Panther applications.
- Write event functions in C or Java.
- Extend the functionality of Panther tools by writing library functions for JPL programmers.
- Write transaction models in C.
- Use Panther tools without JPL.

You can edit Panther's source code and build an executable that incorporates your C functions. You can also install event functions in Panther's source file fun `clist.c`. The full set of Panther library functions is described in *Programming Guide*.

Improving Performance

Consider some of the following possibilities for improving application performance:

- Link objects into the executable.
- Public JPL on the client and/or server.
- Cache service components on the server.
- Minimize the number of service requests.
- Move application logic from the client to the server or vice versa.
- Write C routines or Java methods for common or expensive functions.
- Reduce logging and per service processing on the server.
- Fine tune queries (transaction manager events) or write your own SQL.

Deploying the Application

Once you are ready to deploy the application, you will probably rebuild executables to meet your deployment requirements.

When the process of developing your application is complete, you can easily gather it together, since all components—screens, reports, JPL modules, menus, and bitmaps—are in libraries.

For information on building application executables, refer to Chapter 42, “Building Application Executables.” For information on deploying Panther applications, refer to Chapter 43, “Preparing Applications for Release.”

Part I Preparing for Development

This section is for the senior developer who, in conjunction with the application architect and database administrator, will be setting up the development environment for the application development team and defining the project requirements.

Tasks associated with setting up a development environment are:

CHAPTER

2 Understanding the Panther Distribution

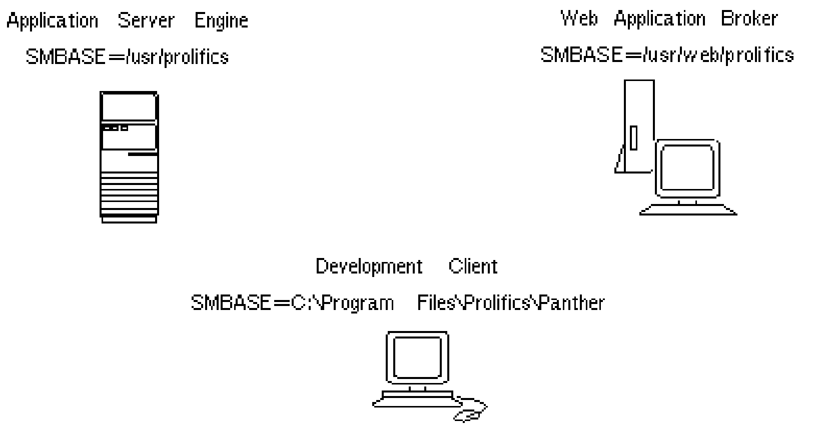
The directory into which Panther software tools are installed is called the Panther installation directory and set in the machine's environment as `SMBASE`.

After installation, your Panther distribution can consist of the following directories:

- `bin`—for JetNet, administrative executables; for COM, DLLs for COM component operations
- `comlink`—for COM, files to rebuild the template `DLL`
- `config`— configuration files
- `docs`—Panther online documentation
- `include`—header files
- `jdb`— JDB libraries
- `lib`—Panther libraries
- `licenses`—license files
- `link`— makefiles for recompiling the executables
- `locale`—for JetNet distributions, localization of system messages
- `nls`—files for Native Language Support

- `notes`—release notes
- `samples`— sample applications and the tutorial's setup files
- `udataobj`— JetNet required `files`
- `util`—Panther executables and utilities. For a more complete list of files in each directory, see the packlist.txt file located in the notes directory.

If you install Panther software on different machines, it is possible for each machine to have its own value for `SMBASE`. When you set the Panther installation directory for applications, be aware of which value is needed.


Application Server Engine
SMBASE=/usr/prolifics
Web Application Broker
SMBASE=/usr/web/prolifics
Development Client
SMBASE=C:\Program Files\Prolifics\Panther

CHAPTER

3 Defining the Project Requirements

Before building the application, the architect in your development team needs to consider the project requirements. With Panther, you can build two-tier or three-tier applications and deploy those applications in a variety of environments. You need to choose the best environment and structure for each application.

Following are questions you should consider.

What Is the Application Architecture—Two-tier or Three-tier?

Two-tier applications are useful for smaller-scale applications. Client workstations contain the presentation and much of the business logic of the application; they have a direct connection to a database server to provide them with data.

The disadvantages of two-tier applications are:

- A client workstation takes up a constant connection to the database even though it only needs that connection a small percentage of the time.
- Users need to update some of the same data concurrently. Depending on the version of database locking, a user will have to wait for the database to release the locks or will have to process the update a second time because the version of data they selected is no longer available.

Three-tier applications address these problems by having three levels: presentation, application logic, and data. Client workstations contain the presentation logic and focus on user interaction. They send service requests to the application server. The application server maintains the connection to the database server and is able to process service requests from multiple clients.

Another advantage of three-tier applications is that the server components can be replicated to run on multiple machines simultaneously. This distributes the client load across multiple machines.

For more discussion of application architecture, refer to Chapter 4, “Defining Application Architecture.”

For Three-tier Applications, Which Middleware Meets Your Requirements?

JetNet and Oracle Tuxedo are TP monitor systems using messages and buffers to pass information to and from clients and application servers. You create client screens and service components in the editor, write the services in C or JPL, and install the services in the JIF. The application server's bulletin board advertises which services are available. JetNet and Oracle Tuxedo are available on UNIX and Windows systems.

MTS is a component-based middleware using components that conform to Microsoft's COM (Component Object Model) specifications. A COM component can support multiple interfaces; each interface is a collection of methods. When you create a COM component in Panther, you define its methods, the parameters expected by those methods, and its properties. In addition, Panther assigns the component a CLSID, a globally unique identifier, and creates a Panther service component which is then installed on the application server. MTS is available on Windows systems.

Panther for IBM WebSphere can be used to build Enterprise JavaBeans which can be deployed on WebSphere Application Server. Enterprise JavaBeans are server-side components written in Java that perform the business logic of an application. Like COM components, Enterprise JavaBeans have a defined public interface of methods and properties that application clients can use to interact with the components. Being written in Java, they are platform independent.

Will the Application Be Deployed on the Web?

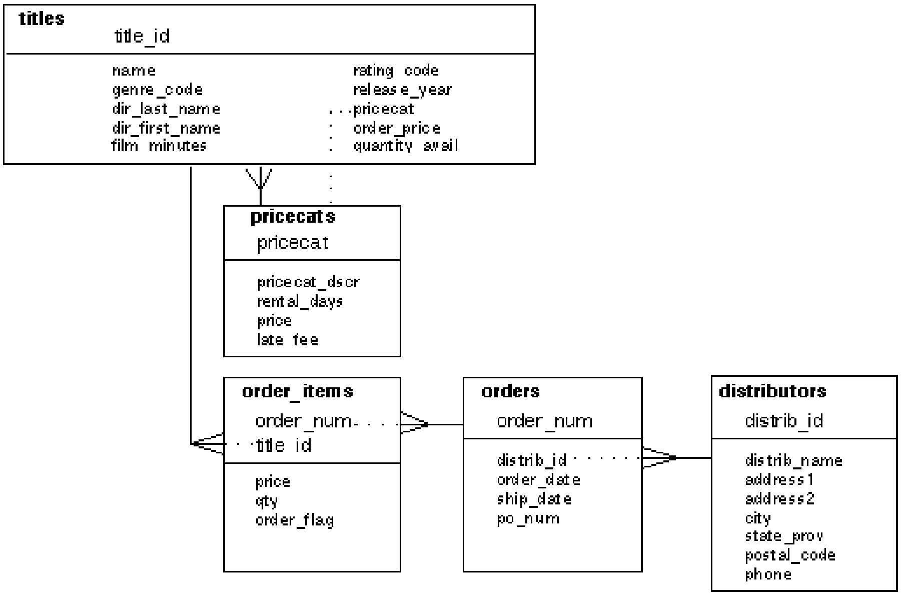
Since the web was originally a text-based delivery system, HTML only dealt with a few widget types and did not contain mechanisms for positioning those widgets. Screens designed in a GUI environment, such as Windows, will appear slightly different on the web.

The application processing also needs to be handled differently since web applications, by default, are stateless. For its web applications, Panther provides mechanisms for saving application state information for each user of a web application.

What Is the Best Database Schema for this Application?

Database diagrams illustrate the structure of the database tables, the primary key columns for each table, and the relationships and foreign key relationships between database tables. By examining the diagram, you can see if the database is normalized—whether the data is only entered once—in the database tables.

The following diagram illustrates a portion of the `vbizplus` database that was used as a basis for some sample reports.



Is a Database Based on that Schema Currently Available?

The development steps outlined in this manual suggest having a database available when you start development. This database is used to populate the repository with repository entries corresponding to the database tables and widgets on each entry corresponding to the database columns.

How Will the Repository Be Used During Application Development?

If a database is used to populate the repository, the database tables and columns are represented in the repository. In addition, you can store screen templates and widget templates in the repository and use those templates throughout the application.

Objects copied from the repository, manually or by using the wizards, inherit the properties of the parent object in the repository. If you change the properties of the parent object, the properties of the child objects can get updated automatically.

How Will the Database Be Accessed?

Panther contains the following software components for database access:

- Middleware API (three-tier only)—Passes service requests through the middleware to the application server.
- Transaction manager—A high level command interpreter that determines what SQL or services must be generated and executed, and asks the next level to do the work.
- SQL generator—Constructs SQL statements and asks the next level to execute them.
- Database interface—Passes SQL requests or stored procedures to the database, and returns formatted results to Panther. The database interface is implemented via the `dbms` verb in the JPL language or the library function `dm_dbms`.
- Database API— Provided by the database vendor.

You can write your own SQL statements, call stored procedures in your database, or have the transaction manager generate SQL statements. In two-tier applications, the client workstation has a direct connection to the database. In three-tier and web applications, the application server maintains the database connections.

Although database access via the transaction manager is usually easiest, you can use any combination in your application. For example, you might allow the transaction manager to handle some access itself, but supply specific SQL statements (for example, stored procedure or RPC calls) for some operations.

If you are upgrading a two-tier application to be a three-tier or a web application, you need to be aware of how database transactions differ in those environments. Since clients running the application do not have a direct database connection, database transactions must be rolled back or committed as part of the service request in three-tier applications or as part of the screen submission processing in web applications.

How Will Service Components Be Used in Your Application?

You can use components:

- To implement all of your database access and business logic.
- To implement repeated tasks in a portion of your application.

One way to organize components in your application is to have one group of components that correspond to database tables and another group of components that implement the business logic. A component in the business logic group could access several database components as it completes a task.

What Are the Hardware Platforms for the Application Clients?

If the client screens run on only one hardware platform, you know that the screen will look the same during development and deployment (provided you use that platform during development).

If the application's screens are to run on multiple platforms, consider which platform is the hardest to please. Usually, it is character mode, followed by Windows, then followed by Motif.

How Computer Literate Is the Application Audience?

If the target audience is the general population, the user interface must be as simple as possible. If the target audience is more knowledgeable, you can design a more powerful and intricate system.

In What Language Will the Programming Code Be Written?

Within the Panther framework, you can code your application in a variety of languages. Panther has its own scripting language, JPL, which can be used for runtime processing. You can also include your own C, C++, or Java code or, for web applications, JavaScript and VBScript functions. You can use any of these languages as you see fit.

What Method Will Be Used for Handling Errors?

Errors that occur remotely may have no meaning to a user, but in most cases, you want to determine which errors should be posted or broadcast from server to client, and what information should be displayed that will be most useful to a user of your application.

During the development process, you will also need a process for logging and updating application problems.

What Type of Network Access Is Available?

The size and speed of the network available when the application is deployed affects how you divide the work in the different tiers.

It is recommended that you devise a prototype going across all tiers of the application to test the network setup and estimate network traffic.

How Will the Work Be Distributed Among the Development Team?

The following development tasks can be assigned to people on the development team:

- Designing the application flow.
- Writing SQL statements and/or stored procedures.
- Programming in C, C++, Java, JPL, JavaScript, or VBScript.
- Designing the user interface/client screens.
- Designing the reports.
- Creating the service components and defining their properties and methods.
- Testing the network access.

CHAPTER

4 Defining Application Architecture

With Panther, you can start with a simple, client/server, two-tier architecture application and when your application requirements increase, convert your application to three-tier architecture. Or, begin with an distributed application structure—create both the client and server components at the same time.

Components of a Panther Application

A typical three-tier Panther application, consisting of the following Panther tools and application components, is illustrated in Figure 4-1.

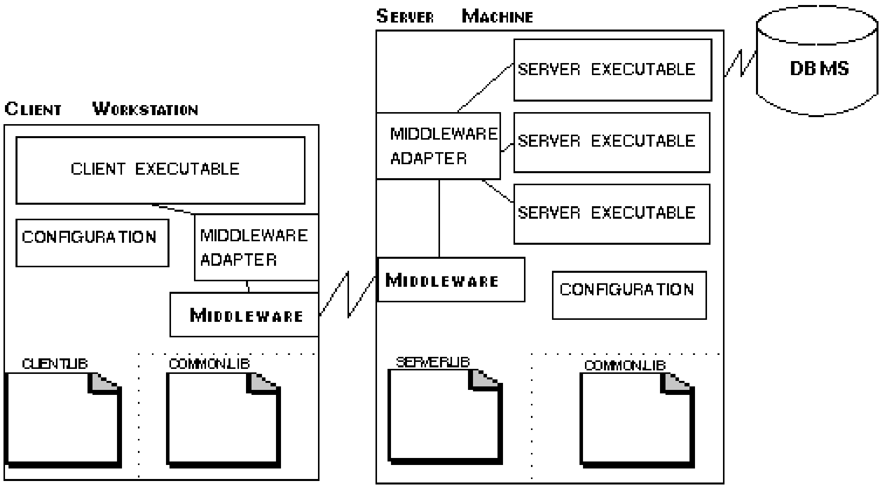


Figure 4-1 Components of an enhanced, three-tier client/server Panther application.

- A database—JDB is Panther's built-in prototyping database which you can use to get started. Interfaces to other database engines are available; one or more of these interfaces is included in each Panther distribution.
- A client executable—Panther provides both basic development and production executables for your client platforms.
- Distributed application environment—The middleware and middleware adapters control three-tier processes and communication between client and application server components.

  JetNet is Panther's middleware product; middleware adapters are also available for Oracle Tuxedo and WebSphere Application Server. On Windows, you can deploy COM components in your Panther application using COM, DCOM, and MTS.
- Server executables to support three-tier architecture—Panther provides the server executables you need for your choice of middleware.
- Client library—Contains the application components that make up the user interface portion of your application. It can store screens, reports, service

components, menus, and JPL modules that you create using the editor as well as any other elements, such as bitmaps, that the interface uses.

- Server library—Contains the service components of a distributed application and their related JPL modules.
- Configuration files—Reside on client machines and, in a three-tier application on server machines, and are used by Panther to interpret your environment.

A three-tier JetNet/Oracle Tuxedo application also has:

- Common library—Contains application-wide objects, such as the JIF, which is the binary file you update to define your application's services, plus any code or objects shared by clients and servers.

Some of these components also exist in simpler, two-tier applications: the database, client executables, client library, and configuration files.

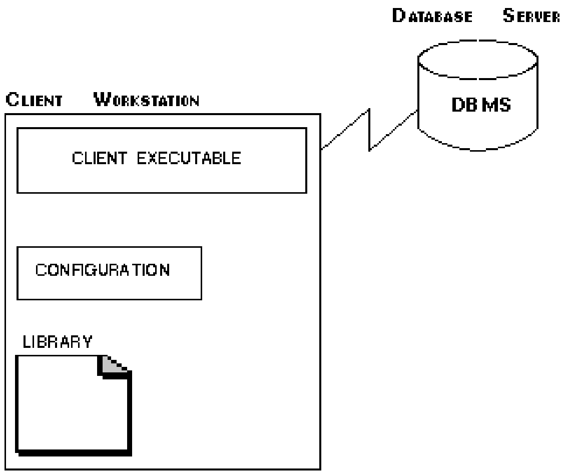


Figure 4-2 Components of a two-tier client/server Panther application.

Building Two-Tier Applications

A two-tier application is distributed between client machine and the database server. The client machine controls the presentation logic and all or most of the application logic. The client can access data in the database through SQL interactions (coded or application-generated) or by making calls to the database's stored procedures.

To build a two-tier application:

- Create an application directory on the client machine.
- Set up the client. Modify `prol5w32.ini`, `prol5w64.ini` or the `Prolifics` resource file and the environment settings as needed to recognize the installation and locate files needed by Panther.
- (JetNet only) If you are providing shared access to remote libraries and repositories, set up a file access server (refer to JetNet Guide/Oracle Tuxedo Guide for details on configuring a server), and create an application directory on the server machine.
- Invoke the editor (`prodev/prodev32.exe`), and create a repository (default name is `data.dic`).
- From within the editor, connect to the database, and import database definitions into the repository.
- Build the application screens using the screen wizard or editor.
- Add programming logic to the screens.
- Add additional application components, such as tool bars, menus, and reports.
- Integrate application components; make the screens work with other screens or reports using push buttons, control strings, and function keys.
- Test and debug the screens within the authoring environment.
- Performance tune.

- Package and deploy the application.

Building Distributed Applications

Distributed applications separate the business logic of the application from the user interface. The application logic is built into service components that reside on the application or component server. The client workstations provide the presentation interface, some application logic, and can make requests to the application servers to perform commonly used logic and access a database. Using service requests, there is no need for a continuous connection between the client and the database server.

Building a JetNet/Oracle Tuxedo Application

The following steps summarize the process for JetNet/Oracle Tuxedo applications:

To build a three-tier JetNet/Oracle Tuxedo application:

- Create an application directory on the Panther application server.
- Create or copy the necessary libraries, such as `server.lib`, `client.lib` and `common.lib`.
- Set up your application server or servers (refer to *JetNet/Oracle Tuxedo Guide*).
- Set up the client. Modify `prol5w32.ini`, `prol5w64.ini` or the `Prolifics` resource file and the environment settings as needed to recognize the installation and locate files needed by Panther.
- Invoke the editor (`prodev/prodev32.exe`), and create a repository that resides on the server machine (default name is `data.dic`).
- From within the editor, connect to the database, and import database definitions into the repository.

- Build client screens, reports, and/or service components using the editor or the wizards. Save objects to their appropriate libraries (on the server machine for shared access and in local libraries for nonshared access).
- Update the JIF with service definitions.
- Add client and server processing; add logic for service components and for client screens. Implement C, Java, or JPL code as needed; for C functions, rebuild executables to incorporate functions.
- Add additional application components, such as tool bars, menus, and reports.
- Integrate application components; for example, make client screens work with other screens or reports by implementing push buttons, control strings, and function keys.
- Test and debug both the client and server portions of the application.
- Performance tune your application to improve event processing.
- Package and deploy the finished application.

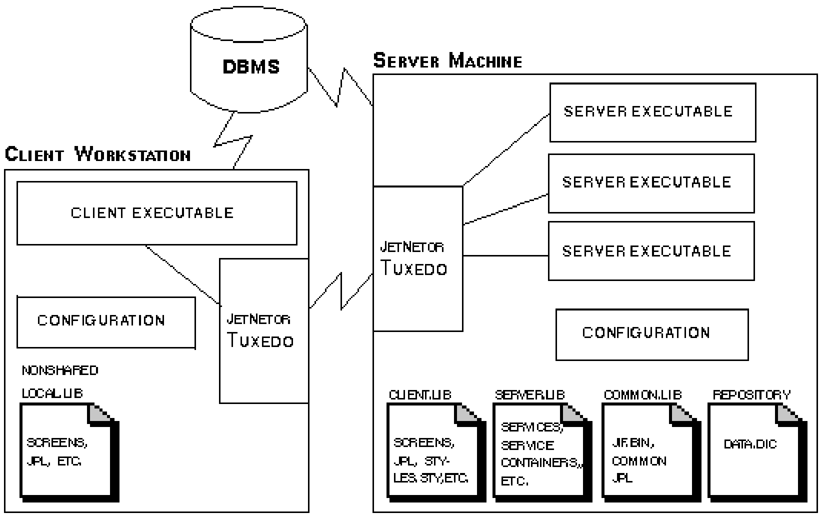


Figure 4-3 Components of a Panther JetNet/Oracle Tuxedo development environment includes access to local and remote libraries as well as direct and remote access to the database.

Building a Component-based Application

The following steps summarize the process for component-based applications:

To build a component-based application:

- Set up the client. Modify `prol5w32.ini` and the environment settings as needed to recognize the installation and locate files needed by Panther.
- Invoke the editor (`prodev32.exe`), and create a repository (default name is `data.dic`).
- From within the editor, connect to the database, and import database definitions into the repository.
- Build client screens, reports, and/or service components using the editor or the wizards.
- Add client and server processing. Add business logic to service components; add user interface functionality to client screens. Implement C, Java, or JPL code as needed; for C functions, rebuild client executables to incorporate functions.
- Save objects to their appropriate libraries (to client.lib for client screens and to server.lib for service components). Saving service components will generate the DLLs and type library files needed for COM components or the Java files needed for EJBs.
- Add additional application components, such as tool bars, menus, and reports.
- Integrate application components; for example, make client screens work with other screens or reports by implementing push buttons, control strings, and function keys.
- Create an application directory on the application server machine (or the machine acting as the COM component server).
- Create or copy the necessary application libraries, such as server.lib, and the component's files to the server machine. For COM components, you need to copy the DLLS and type libraries. For EJBs, you need to copy the jar file containing the Java class files, interfaces, and deployment descriptor.

- Based on your deployment needs, install the components. For COM components, you need to register the components on client machines.
- Test and debug both the client and server portions of the application.
- Performance tune your application to improve event processing.
- Package and deploy the finished application.

CHAPTER

5 Preparing the Application Server

In three-tier applications, the application server:

- Contains the application logic segmented into services or business components.
- Makes services and business components available to the application clients.
- Connects to the data resources needed by the application.

The chapter summarizes the preparation steps needed for:

- JetNet Application Server (page 5-2)
- Oracle Tuxedo Application Server (page 5-4)
- WebSphere Application Server (page 5-4)

In COM/MTS applications, distributed application processing is achieved by deploying COM components and their Panther service components on remote machines using DCOM or MTS. Refer to Chapter 5, “Deploying COM Components” in *COM/MTS Guide* for information.

JetNet Application Server

For the JetNet middleware adapter, the Panther application server can run on a Windows or UNIX machine. To prepare the application server for each application, you need to create the application directory, populate it with the necessary files, apply your environment settings, and configure the types of servers needed for the application.

Create the Application Directory

Create the application directory and populate it with the necessary files.

- For UNIX servers, a setup file with the location of the Panther installation, the license file, and the middleware configuration file at your site. For the default setup file, copy setup.sh from the `config` directory of your Panther installation.
- Three standard application libraries: `client.lib`, `server.lib` and `common.lib`. Copies of these libraries are in the `samples/newapp` directory of your Panther server installation.
- Three standard environment files:
  - `machine.env` for machine-specific settings used for all servers.
  - `proserv.env` for the standard server.
  - `progserv.env` for the conversion server (only used with applications converted from two-tier).

  Copies of these files are in the `samples/newapp` directory of your Panther server installation.
- The middleware configuration file. To create a middleware configuration file, use JetMan or the utility `rbconfig`, and record the settings for `SMRBHOST`, `SMRBPORT`, and/or `SMRBCONFIG`.

Configure the Application Servers

Panther provides you with the ability to configure three types of servers: standard servers for three-tier development and production environments, a conversion server to service three-tier applications you converted from a two-tier architecture, and file access servers used for both two- and three-tier development to provide the development team access to remote libraries, repositories, and report generation facilities.

The application servers in Panther for JetNet:

- Are designed to access built-in event handlers to accommodate and facilitate development and productions needs.
- Offer debuggable capabilities for development.
- Automatically connect to a database.

An application server invokes an executable that contains one or more services that can be called by clients and other servers. Server executables are provided and can easily be initialized using the JetNet manager provided with Panther.

As you develop your application you might find that you'll need to build new executables to incorporate your own C functions, or configure a standard server for production purposes as opposed to development.

See Also

For additional information on configuring the application directory and the server environment, refer to Chapter 2, “Setting the Enterprise Environment,” in *JetNet/Oracle Tuxedo Guide*.

The tutorial also contains a lesson on setting up the server environment (refer to Lesson 2 in *Getting Started-JetNet*).

Oracle Tuxedo Application Server

For the Oracle Tuxedo middleware adapter, refer to Chapter 2, “Setting the Enterprise Environment,” in *JetNet Guide/Oracle Tuxedo Guide* and follow the directions provided with Oracle Tuxedo for setting up and initializing servers.

WebSphere Application Server

WebSphere Application Server provides a full-featured distributed application environment for component-based systems using Java-based technologies. Enterprise JavaBeans created in the Panther editor can be deployed on WebSphere in both web and GUI environments.

For complete information on setting up your WebSphere application server, refer to “How to Set Up the Application Server Engine” on page 2-1 in *Panther for IBM WebSphere Developer's Studio.*

CHAPTER

6 Preparing the Development Clients

In both two-tier and three-tier applications, you need to prepare your client workstations for development. This chapter describes the steps and the changes you can make to the default Panther settings for your project.

For information on setting up web application servers (which operate as Panther clients in a three-tier environment), refer to Chapter 2, “Web Application Setup,” in *Web Development Guide*.

For information about setting up development clients using Panther for IBM WebSphere, refer to “How to Set Up the Development Client” on page 2-8 in *Panther for IBM WebSphere Developer's Studio*.

Copy Your Panther Distribution

You need to make a copy of your Panther distribution for each project. This has the following benefits:

- You can make modifications to the contents of the distribution without modifying the original files.
- At any point, you can compare the distribution file to the working version of the file to see what modifications you made for the project.
- When you install or re-install Panther, it will not overwrite the project-specific versions of files.

For a description of the directory structure in a Panther distribution, refer to Chapter 2, “Understanding the Panther Distribution.”

Configure Your Panther Application

To configure your Panther application, you need to perform the following steps:

- Specify the environment needed for Panther software components.
- Distribute that environment to your client workstations.
- For JetNet or Oracle Tuxedo applications, configure remote access to libraries and repositories.
- Create an application directory on the client.

Specify Your Panther Environment

Panther uses setup variables to point to the Panther installation, other setup files, and other files required by the application, such as libraries. Each developer must set the following setup variable either in their environment or in the Windows initialization file for Panther:

`SMBASE`

(mandatory) The directory path to your Panther distribution. This setting is used to locate Panther software components, particularly `smvars.bin`, the file containing your configuration settings, in the config subdirectory of this location. If `smvars.bin` is not located at `$SMBASE/config/smvars.bin`, then you also need to set the variable `SMVARS`.

Other common variables are:

`SMSETUP`
: Project-specific variable settings.

`SMPATH`
: Directories to search for application files.

`SMFLIBS`
: The libraries to open when you start Panther.

`SMTERM`
: The video driver to use with Panther.

`SMUSER`
: The user name to use in development processes.

Configure Your Project Requirements

You can change other configuration settings as needed by your project:

Colors

Colors in your Panther application can be set by:

- Changing the color properties of objects in the Editor.
- Changing the color scheme of the GUI desktop. For Windows, the Display section of the Control Panel has an Appearance card which sets the color scheme. Versions of X Windows using the Common Desktop Environment have desktop settings.
- Changing the settings of Panther components in the configuration map file. In GUI environments, the default settings match the color scheme of the GUI desktop.
- Defining extended colors in the configuration map file. Once defined, the names will appear in the Extended Color property in the editor and can be used to define colors in Panther components in the configuration map file.

- Changing the color palette settings for the sixteen basic colors in the Motif resource file or Windows initialization file.

The name of the configuration map file varies for the different environments:

- Windows: `wincmap`
- X Windows: `xwincmap`
- Character-mode: `bwcmap, clrcmap`
- Web: `webcmap`

Run `cmap2bin` after making any changes to these files.

Message File

Panther messages are defined in `msgfile.bin`. (The ASCII equivalent is `msgfile`.). Refer to “Using Message Files,” on page 45-2 in *Application Development Guide* for details about message files.

Multiple Platforms

If you are going to deploy your application on multiple platforms, such as Windows and character-mode, you need to be aware of the following settings.

- Colors—Character mode applications have a maximum of sixteen colors—eight basic colors and their highlighted equivalents—while GUI platforms have a much larger color palette.
- Fonts—Character mode applications use a fixed width font while GUI applications generally use a proportional font. With multiple platforms, you will want to coordinate the font settings.

Programming Functionality

If you want to write your own C functions, you will need access to a C compiler so that you can rebuild your Panther executable after installing the functions.

To use Java methods for event processing, you must have access to the JDK during development. When the application is deployed, the application clients must have access to the JRE. The location of the Panther classes (`$SMBASE/config/pro5.jar`) is automatically added at runtime; however, you must specify the location of your own class files in the `CLASSPATH` environment setting.

Remote Library Access

In JetNet and Oracle Tuxedo environments, you can have remote access for libraries and the repository. This allows for greater conformity during application development.

Distribute the Setup to the Client Workstations

In a UNIX environment, you need to distribute the environment settings and create symbolic links or scripts in order to run the executables for each local client. For JetNet/Oracle Tuxedo applications, there are setup files available in `$SMBASE/samples/newapp`.

In a Windows environment where each developer can have a separate copy of the programs, you need to devise a method for each developer to access the master copy of the application files.

Create an Application Directory

It is suggested that you create an application directory for the project on the client, the web application server, and the application server (for JetNet, Oracle Tuxedo and WebSphere). This provides a default location for all your libraries and external programming code (C, Java, JPL).

CHAPTER

7 Initializing the Database

Before your application can access data stored in your database, you must connect to an initialized database engine. Database initialization tells Panther which database driver and which support routine to use to access the database engine.

There are two types of initialization:

- Static initialization—Compiles the database driver and the DBMS interface libraries into the Panther executables. This is the preferred method when developing an application and is generally performed as a part of installing Panther.
- Dynamic initialization—The application loads the support routines at runtime (for example, via the initialization file).

This chapter describes how to implement both initialization types as well as how to initialize one or more database engines.

For information about connecting to a database once it is initialized, refer to Chapter 8, “Connecting to Databases.”

Initializing One or More Engines

A database engine is a DBMS (Database Management System) product. It is identified by a specific vendor and version. For example, SYBASE 10, ORACLE 6.0, and ORACLE 7.0 are three distinct engines. A *Panther database driver* is a C library or DLL that handles all communication between Panther and the DBMS. A *Panther support routine* is the name of the main entry point, or function, in a Panther database driver.

By default, Panther is distributed with a driver for Panther's database engine, JDB, and for additional database engines. On request, drivers are available for other database engines.

Refer to "Database Drivers" for specific information about each Panther database driver.

A Panther application can access zero or more database engines. The application must have a driver for each engine, and it must initialize the engine before declaring a connection.

Initialization Procedure

As a part of initialization, Panther calls the support routine for information on the particular DBMS. The information provides:

- Engine capabilities, for example, whether it can execute stored procedures or support multiple connections.
- Formatting requirements for character, date, and null strings when passed from Panther to the database.
- Default for case handling.

In addition, Panther sets up some structures at initialization, including structures for tracking the number and names of all connections to an engine.

Setting the Default Engine

An application with two or more initialized engines sets the default engine with the command:

```
DBMS ENGINE engineName
```

or sets the *current engine* for a statement by including the `WITH ENGINE` clause. If an application initializes more than one engine, it must set the default or current engine when declaring connections to different engines. Once a connection is declared, the default connection determines the default engine.

Initializing the Database via the Executable

In static initialization, an application identifies the support routines it will use at compile time, and it links with both the Panther database driver libraries and the DBMS interface libraries.

A list of the support routines available for your application is included in the module `dbiinit.c`, which is automatically created from settings found in the `makevars` file for your database when you build Panther executables. When you run Panther, the function `dm_init` is called for each support routine listed in dbiinit.c.

When initialization is successful, the support routine returns zero. If the support routine rejects the initialization and returns an error code, it might be because there is insufficient memory, the engine might not be installed, or the application might have initialized the same support routine more than once. If such an error occurs when executing the Panther initialization routines, `jmain` or `jxmain`, an error message is displayed and Panther terminates.

If necessary, you can create a new version of `dbiinit.c`. For more information on building a new executable and changing static initialization, refer to page 7-6, "Changing Static Initialization."

Database Interface Initialization Routine

The file `dbiinit.c` contains:

- A function declaration for one or more support routines and a corresponding transaction model for your specific database engine.
- A list of engines to initialize in the structure `vendor_list`.
- A list of default transaction models in the structure `pfuncs`.

A sample `vendor_list` structure appears as follows:

```
static vendor_t vendor_list[] =
{
/*  {"engineName", supportRoutine, caseFlag, (char *) 0}, */
    {"jdb", dm_jdbsup, DM_DEFAULT_CASE, (char *) 0},
    { 0 }
};
```

`engineName`
: Can contain any character string that is not a keyword; and is case-sensitive. If an application uses two or more database engines, the *`engineName`* specification tells Panther which database engine to use. Most of the examples in this guide use a vendor name as the *`engineName`* mnemonic, for example `sybase` or `oracle`.

`supportRoutine`
: Is usually in the form `dm_`*`vendorCode`*`sup` where *`vendorCode`* is an abbreviated vendor name. For example:

  - `dm_orasup` for ORACLE
  - `dm_sybsup` for SYBASE
  - `dm_infsup` for Informix

`caseFlag`
: Determines how Panther uses case when mapping column names to Panther variables for `SQL SELECT` statements. Variables can be widgets on the screen, JPL variables, or LDB variables. Using the case setting lets you create all your Panther variables in a particular case and Panther does the conversion to the desired case for you. *`caseFlag`* values are described in Table 7-1.

Table 7-1 *`caseFlag`* **options**

caseFlag	Option	Description
`DM_PRESERVE_CASE`	`p(reserve)`	No conversion is done; uses the case returned by the engine. Column names must match the Panther variable names.
`DM_FORCE_TO_LOWER_CASE`	`l(ower)`	Force column names returned by engine to lower case. Use lower case for Panther variable names.
`DM_FORCE_TO_UPPER_CASE`	`u(pper)`	Force column names returned by engine to upper case. Use upper case for Panther variable names.
`DM_DEFAULT_CASE`	`d(efault)`	Use the default value specified by the Panther support routine. Refer to “Database Drivers” to find the value for a specific engine.

ORACLE, for example, returns column names in upper case.

- If the case flag is set to `DM_PRESERVE_CASE`, the application needs Panther variables with upper case names.
- To map columns to Panther variables with lower case names, set the case flag to `DM_FORCE_TO_LOWER_CASE`.

SYBASE, on the other hand, is case sensitive and can return column names in upper, lower, or mixed cases. To map SYBASE columns to single case Panther variables, set the case flag to either `DM_FORCE_TO_UPPER_CASE` or `DM_FORCE_TO_LOWER_CASE`.

`char`

The last argument, `(char *)0`, is provided for future use.

Changing Static Initialization

When you compile a new executable, the `mkinit` command is executed which creates a new version of `dbiinit.c`, if one does not already exist.

How to Create a New Version of dbiinit.c

1. Delete the current version in your application directory.
2. Edit the database engine settings in the `makevars.`*`dbs`* file (where *`dbs`* is a three-letter abbreviation for your database). Provide the appropriate arguments:

   ```
   dbs_INIT={ d | n | l | u | p }
   ```

   You need to enter one of the command flags. Refer to "Database Drivers" to find the correct dbs abbreviation string; an invalid abbreviation results in an unresolved external error when you try to link.
3. Run `make` (or `nmake` in Windows).

For details on creating new executables, refer to Chapter 42, "Building Application Executables."

Options and Arguments

Most of the command flags deal with the case conversion for column names. When a query retrieves a column value, Panther looks for a Panther variable with the same name as the column name and places the value in that variable. However, some database engines only create column names in a specific case or allow mixed cases. Using the case setting, you can create all your Panther variables in a particular case and Panther handles the conversion for you.

`d`

Use the default case conversion set in the support routine. Refer to "Database Drivers" to find the setting for a particular database engine.

`n`

Do not install this engine in `dbiinit.c`.

`l`

Convert column names to lower case.

`u`

Convert column names to upper case.

`p`

Preserve the case of the column names when converting to Panther variable names.

`dbs`

Three-letter abbreviation that Panther specifies for the database. For example, `jdb` is for JDB and `syb` for SYBASE.

Refer to “Database Drivers” to find the abbreviation for a particular database engine.

Dynamic Database Initialization

In dynamic initialization, the application identifies the desired support routines at runtime. No compilation is needed to change the initialization.

Under Windows, you have two methods of initializing a database engine. One method is to compile the database into the executable (refer to “Changing Static Initialization” on page 7-6 for details on static initialization); the other method uses specifications in your Panther initialization file (`PROL5W32.INI` or `PROL5W64.INI`) to initialize database engines at runtime when the Panther program is started.

How to Identify the Database Engines in the Initialization File

Add a database-specific section to the file. The syntax is:

```
[databases]
installed = engineName [engineName]
default = engineName

[dbms engineName]
case={upper | lower | preserve | default}
driver=driver DLL
model=model DLL
```

`engineName`

(Required) The name assigned to the database engine. It can contain any character string that is not a keyword; it is case-sensitive. If an application uses two or more database engines, the `engineName` specification tells Panther which database engine to use. Most of the examples in this guide use a vendor name as the `engineName` mnemonic, for example `sybase` or `oracle`.

The default `engineName` specification is optional and lets you name a default database engine. If there is no specification, JDB is the default database engine.

`case`

(Optional) Determines how Panther uses case when mapping column names to Panther variables for `SQL SELECT` statements. Variables can be widgets on the screen, JPL variables, or LDB variables. Using the case setting lets you can create all your Panther variables in a particular case and Panther converts to that case for you. The case options are described on page 7-5 in Table 7-1.

`driver DLL`

(Required) The DLL provided by Panther for use with a particular database engine and set by the installation program. For additional information about DLLs for a specific engine, refer to the `README.`*`vendorCode`* file in the distributed `NOTES` directory. (*`vendorCode`* is a three-letter abbreviation for the vendor, for example, `syb` for SYBASE).

`model DLL`

The name of the transaction model provided by Panther for use with the transaction manager, and is required if you are using the transaction manager (for example, building screens withe screen wizard) and are not compiling a custom transaction model. For information about the transaction model for a specific engine, refer to the README.vendorCode file in the distributed NOTES directory.

CHAPTER

8 Connecting to Databases

Once the engine is initialized, you need to establish a connection before your application can access any data. There are three ways to connect your application to the database, depending on your specific development requirements, the application architecture, and your application's requirements. You can connect to a database:

- Via server initialization (for JetNet/Oracle Tuxedo applications)—Set the Database Connect String option from the Server Configuration dialog in the JetNet manager.

  (Refer to "Database Connect String" on page 3-26 in *JetNet/Oracle Tuxedo Guide* for details on setting server properties and database connections using the JetNet manager. For Oracle Tuxedo, refer to "Initializing Servers" on page 8-17 in *JetNet/Oracle Tuxedo Guide*).

- Directly through the screen editor (or in test mode).

- Programmatically—Make runtime connections using the `DBMS DECLARE CONNECTION` command in a JPL procedure or C function.

In addition, the database engine must recognize your application users and grant them proper permissions, which can involve changes by your database administrator.

Connecting to the Database in the Screen Editor

While you are designing your application, you need a direct connection to the database server if you are:

- Developing a two-tier application—to import data definitions into a repository and to test your screens using "real" data.
- Developing a three-tier application—to import from the database to a repository and, for JetNet and Oracle Tuxedo applications, to test service components. (Refer to Chapter 38, “Testing Application Components,” for instructions on how to test service components).

Notes: During JetNet and Oracle Tuxedo development, making a direct connection to the database server does not affect a database connection made via server initialization.

How to Make a Direct Connection to the Database from Within the Screen Editor

1. Choose File→Open→Database or in test mode, choose Database→Connection. The Choose Engine dialog opens.
2. Select the desired engine and enter a connection name, if the default name is not appropriate.
3. For some engines, a Connect to Database dialog opens where you can enter connection options. The options vary according to the selected engine. For JDB, an engine-specific Open File dialog is displayed from which you can select the desired JDB database file.

How to Close a Database Connection Within the Screen Editor

Choose File→Close→Database or in test mode, choose Database→Disconnect. Both options close direct connections on a specified engine. They do not close connections that have been established via the application server.

Programmatically Connecting to the Database

A declared connection is a named structure describing a session about an engine. Two-tier applications contain the dbms commands to make direct connections to the database; in three-tier JetNet and Oracle Tuxedo applications, the dbms command is part of the application server initialization.

The syntax for `DBMS DECLARE CONNECTION` is:

```
DBMS [ WITH ENGINE engineName ] \
   DECLARE connectionName CONNECTION WITH \
   OPTION=argument[, OPTION=argument ...]
```

The information provided about the engine includes:

- A connection name—If a `WITH ENGINE` clause is not specified, the connection is declared for the default engine. Connection names specified in the statement are case-sensitive.
- Logon information supplied by the option arguments—Different engines support different options. Common options include `USER`, `PASSWORD`, `DATABASE`, and `SERVER`. If an option is not supported by the engine, the database driver reports an error—Bad arguments. To see a list of options for a specific engine, refer to "Database Drivers."
- Arguments that contain the value assigned to the option. The argument can be a screen variable, a JPL variable or a quoted string.
- A data structure for a default `SELECT` cursor.

- Pointers to other structures associated with the connection, including named cursors (thus when an application closes a connection, Panther is able to close all open cursors on the connection). For more information about database cursors, refer to page 28-3, “Using Database Cursors.”

Once a connection is opened, the application can operate on the database tables.

A sample statement for JDB is:

```
DBMS DECLARE vid_conn CONNECTION WITH DATABASE="videobiz"
```

How to Close Database Connections

At runtime, the application can execute the following command for each declared connection:

```
DBMS CLOSE CONNECTION connectionName
```

```
or
```

```
DBMS CLOSE_ALL_CONNECTIONS
```

Setting Default and Current Connections

A connection is always associated with an initialized engine. Setting a connection as the default or current connection also sets the default or current engine. When using multiple connections, you should set a default connection.

How to Set a Default Connection

Use the following command:

```
DBMS CONNECTION connectionName
```

How to Override a Default Connection

Use a `WITH CONNECTION` clause to specify the connection to use for a single statement. For example

```
DBMS WITH CONNECTION oracon QUERY SELECT * FROM customers
```

Multiple Connections to a Single Engine

Some database engines permit two or more simultaneous connections.

Refer to “Database Drivers” to see if this option is available for your engine.

How to Make Multiple Connections to the Database

Declare a named connection for each session on the engine. (The engine must support this feature.) For example:

```
DBMS ENGINE sybase
DBMS DECLARE s1 CONNECTION WITH \
   USER=:+uname, PASSWORD=:+pword, SERVER='birch'
DBMS DECLARE s2 CONNECTION WITH \
   USER=:+uname, PASSWORD=:+pword, SERVER='maple'
DBMS CONNECTION s1
```

This example declares two connections on the sybase engine and sets the default connection to be `s1`. Panther gets the values for `USER` and `PASSWORD` from the variables *`uname`* and *`pword`* at runtime.

If you execute an additional connection statement for an engine supporting multiple connections, the support routine opens the additional connection and Panther keeps a count of the number of active connections for the engine.

If the engine does not support multiple connections or if the connection name is not unique, Panther returns the error `DM_ALREADY_ON`.

How to Close All Connections on an Engine

Executing the following command:

```
DBMS [ WITH ENGINE engineName ] CLOSE_ALL_CONNECTIONS
```

Connecting to Multiple Engines

If a two-tier application uses two or more database engines, a connection must be declared for each. You can then set a default connection. For example:

```
DBMS WITH ENGINE sybase DECLARE sybcon CONNECTION WITH \
   USER=:+uname, PASSWORD=:+pword, SERVER='birch'
```

```
DBMS WITH ENGINE oracle DECLARE oracon CONNECTION WITH \
   USER=:+uname, PASSWORD=:+pword
DBMS CONNECTION sybcon
DBMS QUERY SELECT * FROM titles WHERE title_id = :+title_id
```

In the example, connections are declared on the engines sybase and oracle. Panther gets the values for `USER` and `PASSWORD` from the variables *uname* and *pword* at runtime. The connection `sybcon` is identified as the default engine. Therefore, Panther performs the `SQL SELECT` on connection `sybcon` and uses the support routine associated with the sybcon engine to execute the query.

In three-tier applications, connections to multiple database engines is usually handled by having a different application server for each database engine.

Checking the Status of Connections

How to Find out If a Database Connection is Open

Check whether a database connection is open using the library function `dm_is_connection`. For example:

```
// This procedure finds out if the connection is
// open and if not, declares the connection.

proc check_connect
vars retcode
retcode=dm_is_connection("app_connect")
if retcode == 0
{
   DBMS DECLARE app_connect CONNECTION WITH ...
}
return
```

How to Find out the Database Connection Assigned to a Database Cursor

Find out which database connection is assigned to a database cursor using the library function `dm_cursor_connection`. For example:

```
// This procedure finds the connection for a cursor
// and makes it the default connection.

proc check_cursor
vars retcode
retcode=dm_cursor_connection("select_data")
DBMS CONNECTION :retcode
return
```

How to Find out the Handles to a Database Connection

In order to interface with some database engine programs, you need information about the Panther database connection structure. Use the library function `dm_get_db_conn_handle` to obtain this information.

Verifying Database Access

Depending on your application architecture, you need to verify that users, processes, and services have database access and appropriate permissions on database tables or views.

UNIX

For two-tier UNIX applications, verify that:

- The user specified in the `DBMS DECLARE CONNECTION` statement is configured for database access.

- The owner of the application process sets the environment variables needed for database access.

For two-tier UNIX web applications, verify that:

- The user specified in the `DBMS DECLARE CONNECTION` statement is configured for database access.
- The web application's initialization file contains the environment variables needed for database access. (At runtime, the web application runs under an http process.)

For three-tier JetNet and Oracle Tuxedo UNIX applications, verify that:

- The user specified in the `DBMS DECLARE CONNECTION` statement (part of the proserv configuration or JPL initialization) is configured for database access.
- The owner of the application process sets the environment variables needed for database access.

Windows

On Windows, you can install applications as a service. Windows assigns that service a user (owner) name; that user name must be configured for database access. Verify that the service has database access by:

- Logging into the machine as the user (owner) of the service.
- Starting a database vendor's program.

Some database engines have special installation instructions. For example, Informix requires that you run `setnet32` to add the service user and then run the demo login program to check that the user was added correctly.

For two-tier Windows applications, verify that:

- The user specified in the `DBMS DECLARE CONNECTION` statement is configured for database access.
- The PC has the database vendor's client program installed.

For two-tier Windows web applications, verify that:

- The user of the web application's service is configured for database access. That user can be specified as part of the monitor command that installs the application as an service.
- The user specified in the `DBMS DECLARE CONNECTION` statement is configured for database access.
- The PC has the database vendor's client program installed.
- The web application's initialization file contains the environment variables needed for database access.

For three-tier JetNet and Oracle Tuxedo Windows applications, the application uses the TUX IPC Helper service. Verify that:

- The user that the TUX IPC Helper service runs under is configured for database access.
- The user specified in the `DBMS DECLARE CONNECTION` statement (part of the proserv configuration or JPL initialization) is configured for database access.

If the three-tier Windows application includes a web application client, the web application can be installed as a service. Verify that:

- The user of the web application's service is configured for database access. That user can be specified as part of the monitor command that installs the application as a service.

CHAPTER

9 Connecting to the Middleware

Panther is a complete three-tier product containing all the client and server components needed in building applications. Clients and servers can reside on different machines connected by a network. The middleware controls communication between client and server components, making it possible for an application's clients to interact with the application's servers.

Panther can work with your choice of the following middleware packages:

- JetNet, Panther's own middleware package.
- Oracle Tuxedo
- MTS, 32 bit Windows only
- WebSphere Application Server

JetNet and Oracle Tuxedo are TP monitor systems using messages and buffers to pass information to and from clients and application servers.

MTS for Windows, one of the deployment options for COM components, controls database connection pooling, transactions and security access for COM component packages.

IBM's WebSphere Application Server deploys Enterprise JavaBeans (EJBs), Java-based components, in a distributed application environment.

This chapter describes:

- What tasks the middleware performs.

- Where the middleware fits into your application.

Using JetNet and Oracle Tuxedo

In JetNet and Oracle Tuxedo applications, the middleware performs the following tasks:

- Establishes client connections to the application. These connections have the option of making use of several client authentication mechanisms. Refer to the `client_init` command for further information on client initialization and refer to "Client Authentication Functions" on page 44-28 for more information on client authentication.

  If you are in the editor, refer to page 9-3, "Opening a Middleware Session in the Editor."

- Forwards service requests to an appropriate server for processing. Making a service call is similar to making a function call, but the function can be executed on a remote machine, and the code making the service call does not need to include any information about the remote machine.

  When no server is immediately available, service requests are queued up for the next available server.

  Service calls can be made either synchronously where further processing is blocked until the reply is received or asynchronously where client processing continues. In the asynchronous case, a reply is received by the client at a later time. Refer to the `service_call` command for detailed information on service calls.

- Returns data and status to a client following a service request.
- On the client, it can instruct the transaction manager to access the services that will carry out the database requests made by the client.
- Allows client and server agents to send messages to other agents. Refer to the broadcast and notify commands for more information.

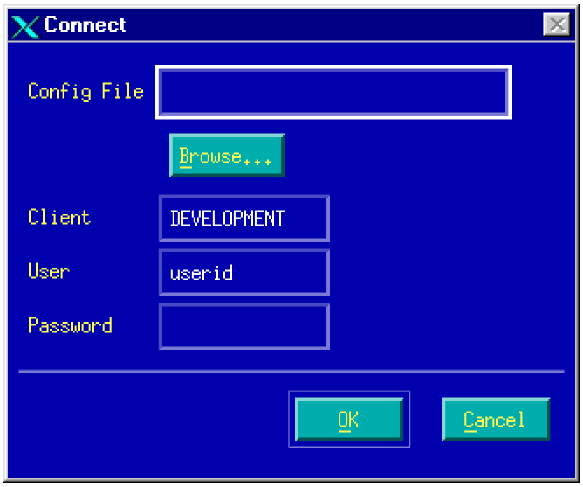
- Permits a server to advertise its services to clients. Refer to the advertise command for more information. Servers can also dynamically change their service offerings.
- Automatically restarts servers if they fail, providing the servers have been configured for automatic restart. For more information, refer to “Server Restart Frequency” on page 3-21 in *JetNet/Oracle Tuxedo Guide*.

Opening a Middleware Session in the Editor

In JetNet/Oracle Tuxedo applications, you open a middleware session to connect to a middleware and access the libraries and repositories on the application server. The middleware controls processes and communication between the application's clients and servers. To test an application screen that uses services, you must have a valid middleware connection.

How to Open a Middleware Session

1. Choose File→Open→Middleware Session. The Connect dialog box appears.



Config File

Client DEVELOPMENT

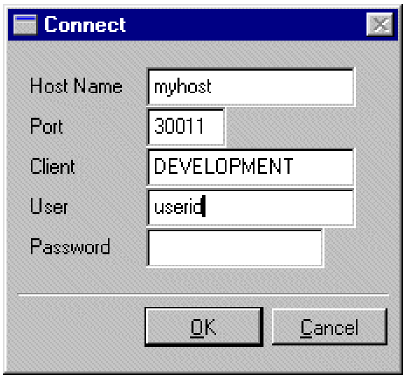
User userid

Password

Figure 9-1 Connect to the middleware for a client on a server machine, also called a local or native client.

For a local client on a server machine, the Connect dialog box has the following fields and specifications:

- Config File—By default, displays the value of `SMRBCONFIG` variable. Otherwise, use the Browse button to select your configuration file.
- Client—Specify client type by name (default is `DEVELOPMENT`). The client name can be no greater than 31 characters.
- User—Specify user account (login) name. The user name can be no greater than 31 characters.
- Password—Specify application password. The password can be no greater than 8 characters.



Host Name: myhost
Port: 30011
Client: DEVELOPMENT
User: userid
Password:

Figure 9-2 Connect to the middleware for a client not on a server machine, also called a remote client.

For a client not on a server machine, a remote client, the Connect dialog box has the following fields and specifications:

- Host Name—By default, displays the value of the `SMRBHOST` variable. This variable provides the network address of the machines to which the client will connect.
- Port—By default, displays the value of the `SMRBPORT` variable. This variable provides the port numbers associated with the machines to which the client will connect.
- Client—Specify client type by name (default is `DEVELOPMENT`).
- User—Specify user account (login) name.
- Password—Specify application password.

2. Enter the information required in the appropriate fields depending on whether you are a client on a server machine or a PC client.

   If the application requires level-two authentication, then you must enter the application password in the Password field.

   Note: The level of authentication is decided at design time. Thus, a user should be aware of the decided authentication level.

Opening a Middleware Session Programmatically

To establish a middleware session programmatically, use the `client_init` command. For example, this statement opens a client connection and specifies an application password:

```
client_init PASSWORD appPassword
```

Refer to `client_init` for more information.

Using MTS

Since a COM component's entry in the Windows registry also specifies its machine location, the client application uses the same processing to call components on local and remote machines.

MTS is just one of the deployment options. COM components can be deployed using the following technologies:

- COM—Component Object Model. For COM applications, you install the component on each application client.
- DCOM—Distributed COM. For DCOM applications, you install the component on one machine and run a registration program on each application client to point to the component's remote location.

- MTS—Microsoft Transaction Server. For MTS applications, you install the component using the Microsoft Management Console and export an installation file to install and register components on an application client.

For detailed information about deploying COM components, refer to Chapter 5, “Deploying COM Components,” in *COM/MTS Guide*.

Using WebSphere Application Server

In WebSphere Application Server, you deploy EJBs on the WebSphere server machine. Clients in your application can then call methods and set properties on those EJBs. Clients in a Panther application can specify the WebSphere server using the `provider_url` application property.

For information on setting up Panther software in a WebSphere environment, refer to Chapter 2, “Configuring Machines,” in *Panther for IBM WebSphere Developer's Studio*.

CHAPTER

10 Accessing Libraries

Panther's libraries and repositories store screens, reports, binary JPL files and other application components. To ensure that all members of your development and design team have access to the same information and sets of standards, you want to allow everyone access to these libraries and repositories. For the purpose of this discussion, the term library describes both libraries and repositories.

The development process often depends on coordinating write-access to these files—allowing access by several people and providing a means of knowing when changes were made.

In addition to providing its own mechanism for controlling multi-user access to libraries and repositories, Panther also provides an interface to source control management systems, specifically SCCS, PVCS and MSSCCI, that let you take advantage of source management systems while in the editor environment.

Panther provides:

- Its own support of multi-user access to libraries that are not under a third-party source control management system.
- Support for your source control management system to maintain libraries and repositories.
- Access to members under source control directly in the screen editor.

Libraries provide a convenient way to distribute a large number of screens with an application, and improves efficiency by eliminating paths searches at runtime and the number of files that are open. You can have multiple Panther libraries open during development and at runtime. As for repositories, while you can create multiple repositories, you can only have one open at a time during an editor session, and its contents should be accessible to the entire development team.

Configuring Your Library Access

Two-tier applications

For two-tier applications, application objects are stored in libraries, such as `client.lib`. To provide access to the entire team, store the libraries on a common file server.

JetNet and Oracle Tuxedo applications

For the JetNet and Oracle Tuxedo middleware adapters, there are three basic libraries: `client.lib`, `common.lib` and `server.lib`.

You can open the appropriate libraries for the application servers, application clients, and developers by setting `SMFLIBS`.

During development, the libraries can be stored on the application server and accessed remotely by the developers, in order to be available to the entire development team. For remote access, the application must be configured with a file access server (`devserv`). For more information on configuring servers, refer to Chapter 3, “Configuring the Enterprise,” in *JetNet/Oracle Tuxedo Guide*.

COM and MTS applications

For COM/MTS applications, a client library (by default `client.lib`) contains the user interface elements of the application. A server library (by default `server.lib`) contains the service components and is generally located in the application directory. For more information, refer to Chapter 5, “Deploying COM Components,” in *COM/MTS Guide*.

Managing Library Access

Accessing Library Members Outside of Source Control

If you do not use or have a source control management system, Panther provides support for controlling multi-user access to screens, reports, menus, and JPL modules in libraries.

Panther's approach to multi-user access is to inform rather than enforce. To prevent inadvertent damage, Panther informs you if another user currently has a "reservation" on (that is, write-access to) a file you are trying to open, save, or delete; but you are given the opportunity to "steal" the reservation.

This method pre-supposes a high degree of communication and cooperation among the members of your development team. For example, a user who initially has a reservation on a screen might later find, when attempting to save or delete it, that another user now has the reservation. The first user can, of course, steal the reservation back, but stealing reservations without adequate communication between users can result in the loss of someone's work.

While Panther provides enough information to prevent inadvertently damaging another's work in progress, it does not enforce file locking. Users are responsible for checking with the holder of a reservation before deciding whether or not to steal it.

If you require a more secure file-locking method to control multi-user access, use one of the third-party source control management systems that Panther supports.

Opening Library Members

Panther warns you if you choose to open a library member (screen, report, JPL module, or menu file) that is being edited (open with read/write privileges) by another user. You can choose to steal the reservation or open the library member with read-only privileges. If you choose Yes to steal the reservation; the requested library member opens with read and write privileges; choose No to open the member with read-only privileges.

A read-only library member is displayed with the last saved changes. If you edit a read-only file, you can use File→Save As to save the library member with a different name.

Closing Library Members

When you choose to close a library member (having read-write privileges) that you have been editing, Panther displays the following message:

```
Do you want to release your reservation of <filename@lib>?
```

Choose Yes to release the reservation; the library member closes and is available for editing (write-access) by other developers.

Choose No to keep the reservation; the library member closes and can be open as read-only by other developers.

Releasing a Reservation

To automatically release a reservation whenever you close library members, choose Options→Auto Release on Screen Close. The message for releasing a reservation no longer displays; the reservation is automatically removed and the closed screen is available with read/write privileges to developers.

Maintaining Libraries Under Source Control

The process of putting a library under source control places the contents—its library members—under source control management. Therefore, you must check library members into a source controlled library (more on check-in later) to actually place the individual members under source control. For the purpose of this discussion, it is assumed that library members or repository entries—screens, reports, JPL modules, or menu files—when checked in, are stored by the source control management system.

Use of source control management systems requires that your path allows you to run the Panther utilities `f2asc`, `jpl2bin` and `m2asc` if you are not storing the screens in binary. These utilities converts the screens, reports, JPL and menus to ASCII before they are checked in. Additionally, users of PVCS must set the `VCSID` and `VCSCFG` variables. For more information, refer to your vendor's documentation.

Before putting a library under source control, you must have a new or populated library (created within the screen editor or by `formlib` using the `-c` option).

How to Provide an Interface to Your Source Control Manager

1. From the command line, type:

   ```
   formlib -g "sourceMgr [-b] mgrArgs" library
   ```

   For example, for SCCS support, use:

   ```
   formlib -g "sccs devdir" dev.lib
   ```

   Or for PVCS, use:

   ```
   formlib -g "pvcs c:\DEV\ARCHIVE" dev.lib
   ```

   Or for MSSCCI (Microsoft Source Code Control Interface), use:

   ```
   formlib -g "scpi Provider='Jazz MSSCCI Provider'
   ProjectName='Panther First Project' LocalProjPath='c:\Panther
   First Project sandbox'" dev.lib
   ```

`sourceMgr`	Name of the installed source control management driver—either `sccs`, `pvcs` or `scpi` (lowercase).
`-b`	Optional flag that causes Panther to store files in the configuration management system in binary.

`mgrArgs`	A string that is used by the `sourceMgr`-specified driver.
	For SCCS, supply the name of the source control directory—either relative to the library's directory or an absolute path—where the application-related files are to be stored.
	For PVCS, supply the name of the archive directory—either relative to the library's directory or an absolute path—where the screens and revised files are to be stored.
	For MSSCCI, a list of tokens and values separated by equal signs. The values can be quoted using single, double or back quotes; otherwise, spaces and backslashes should be escaped with backslashes. The supported tokens are:
	■ `Provider` - required. It is used to find the DLL that provides access to the source control system using the MSSCCI API.
	■ `ProjectName` - used to connect to a project.
	■ `LocalProjPath` - used to specify the base directory where files will be exchanged with the MSSCCI provider. If `ProjectName` and/or `LocalProjPath` is omited, the MSSCCI provider will be asked what they should be. Many providers have a dialog allowing the user to select the needed values. Panther will update `mgrArgs` with the values.
	■ `WorkingDir` - can specify a subdirectory of `LocalProjPath` where the members of this library are to be exchanged with the MSSCCI provider.
	■ `User` - used by some providers to supply the user name. There is no way to supply a password, so that there may be a dialog if a password is needed.
	■ `CallerName` - a name to be used in MSSCCI provider dialogs, The default is `'Panther'`.
	■ `AuxPathLabel`, `AuxProjPath` - usage depends on the MSSCCI provider.

`library`	Name of the Panther library or repository that you are putting under source control management.

2. If the library already contains members, invoke the screen editor so that you can check in screens, reports, JPL and menus to the source control management system:
   - If the library is not open, choose File⟶Open⟶Library or Repository.
   - Open the screens that you want under source control. Choose File⟶Open⟶Screen or Repository Entry.
   - Choose File⟶Source Mgmt⟶Check In. The screen is stored in the specified source directory.
3. If you want to check in a screen from another library:
   - Open the desired screen.
   - Choose File⟶Save As and save the screen to the library that is under source control.

Library Members Under Source Control Management

There are three features of source control management available to you when you open a library whose contents are being maintained under source control management. You can open a copy of a file so that you can edit it or you can open it read-only. You can also check changes in and take full advantage of the features offered by your source control manager to monitor those changes.

The following options are available from within the editor workspace so that you can update and view library members (screen, reports, menus, styles, and JPL modules) by choosing the desired option:

- Check Out—Available by choosing File⟶ ²Source Mgmt. The Check Out option opens a writable copy of the selected library member. Other users trying to access the same library member for edit/update are notified when attempting to edit the library member. Moreover, a library member can be opened read-only when it is checked out by another user.

Panther automatically checks out a library member under source control management when you choose File→Open.

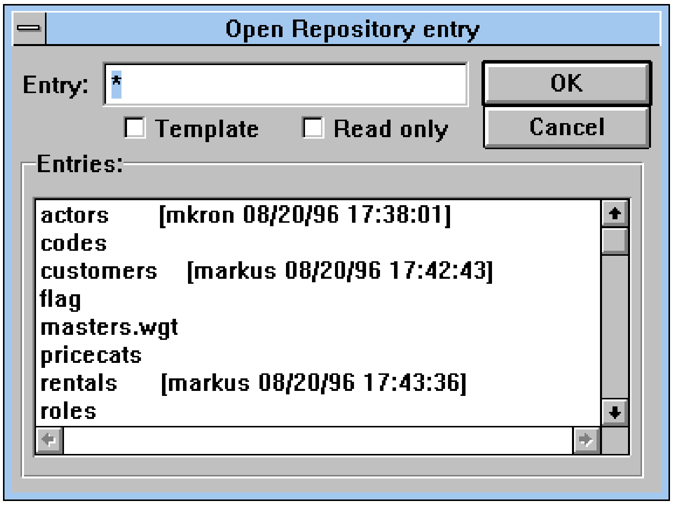
- Check In—Available by choosing File→Source Mgmt. The Check In option specifies that changes to the selected member be checked into the source management directory. If the member is not already checked out, the source control manager determines how to handle the check in request.
- Cancel Check Out—Available by choosing File→ ²Source Mgmt. The Cancel Check Out option cancels the check out request. The library member is made read-only and the lock is released. Another user can check out the same member for edit/update purposes.

Warning: Panther normally converts the library members to ASCII on Check In and from ASCII on Check Out. If binary versions of the library members are stored in the library, instead of ASCII, error messages will result unless the `-b` option is specified in the library's configuration string.

How to Edit a Library Member Under Source Control Management

1. If the library or repository is not open, choose File→Open→Library or Repository.
2. Select the desired library/repository to view its contents.

   All members of the selected library/repository are listed in the appropriate Open dialog. Those members currently checked out are listed along with the name of the user and date and time of check out.



```
actors       [mkron 08/20/96 17:38:01]
codes
customers    [markus 08/20/96 17:42:43]
flag
masters.wgt
pricecats
rentals      [markus 08/20/96 17:43:36]
roles
```

Figure 10-1 Checked out library members are identified on the Open dialog boxes.

Note: Ownership information, that is, the user who has the library member checked out, might not reflect the latest source control management status. This is due to the fact that source control can be executed outside Panther. Panther updates the status once you make a selection.

3. Select the desired member from the list. If multiple revisions of the library member are stored in the source management directory, you can choose which revision to open.

   - If you choose to open a member that is checked out by someone else, a message informs you who the user is and the date and time of check out. You can choose to open the member read-only.
   - If Panther finds a library lock, a message is issued. You can choose to wait for the lock to clear or cancel the check out. While Panther continues to check for the removal of the lock, the message is redisplayed five more times. If you choose to continue, Panther allows you to override the lock.

     In the event that the lock cannot be released, check for read/write permissions on the library and on the directory in which it resides. If that fails, check for a `*.jlk` file in the directory where the library resides.

4. Edit the library member as required.

You can make changes, and save (choose File⟶Save) those changes as often as you need to without checking the member back into source control. As long as the member is checked out to you, other users can only retrieve a read-only copy. Moreover, the read-only copy is created from the last `check_in`; it does not reflect the latest changes saved to the library. Once you are done with the library member, and are satisfied with the edits, you can check it back into the source control management system.

Note: Choosing File⟶Save does not perform a check into source control. You must proceed to step 5 to update the source copy.

5. Choose File⟶Source Mgmt⟶Check In. The library member closes and the lock is released.

   If you made changes since the member was last saved, you are prompted to save those changes to the library:

   - Choose Yes to save to the library. The revised copy is saved to the source control manager, and the source control manager handles the rest of the check-in, such as prompting for comments, and displaying update information.
   - Choose No to check in the last saved copy. Your current changes are not saved to the library or source control management directory.

How to Save a Read-only Library Member and Store It Under Source Control Management

1. Make sure the open repository or selected library is under source control management (converted with the `formlib` utility).
2. Choose the appropriate File⟶Save As option. The Save As dialog box opens.
3. Enter a name for the library member. If you want to save the member with the name as one that already exists, enter the name or select it from the list.
4. If there are multiple revisions of the member in source control, you can choose which revision should get the latest changes. Choose OK.

   Note: Saving a library member does not check it into the source control management system.
5. Choose File⟶Source Mgmt⟶Check In.

How to Delete a Library Member that Is Under Source Control

You can remove the working version of a screen (or other member) from the library under source control, but the source (previous versions) are not removed. To remove all traces of an entry, you must access your source control management system directly (outside of Panther).

Cancelling a Check Out

When you open a library member that is in a library under source control management, a write-lock is enforced on that member. Another user cannot edit the file.

In the event that you do not want to edit the member, or you don't want to save the changes you've made since you checked it out, choose File⟶Source Mgmt⟶Cancel Check Out.

A cancelled check out means that your current edits, even saved edits, are not saved to the source control management system. The library member in source control management remains unchanged. The write-lock is removed and another user can now check out the member.

Note: Choosing File⟶Revert restores a screen to its last saved copy as it exists in the library; it does not restore to the copy that was last checked into the source control management system.

Part III Creating Application Building Blocks

This section describes the building blocks of your Panther application.

CHAPTER

11 Creating and Using a Repository

The visual object repository is used during development to define and store a set of objects needed to build screens, service components, and reports. Once the repository is populated, you can easily make a new application component by copying the necessary objects from the repository.

In addition to the development time saved by creating objects only once, the repository can be used to easily update application objects by using inheritance. When you copy an object from the repository, the copy, or *child*, retains the property definitions of the original object, the *parent*. If you change the properties of the parent object in the repository, the properties that the child has inherited are also updated.

The visual object repository provides:

- A mechanism to assure consistency and control among all application components.
- A single reference for data elements and templates used in an application.
- A facility for propagating database table and column information to your application.
- A single access method for propagating changes to widget and screen properties without having to individually edit each application screen.

About Repositories and Inheritance

A repository can consist of one or more entries; you can create, view, and edit repository entries with the editor. Like Panther libraries, other developers can access the repository concurrently. The repository can provide consistency and control over the look and feel of your application, and its data elements.

By copying an object, such as a text widget, from the repository to your application screen, you create an inheritance relationship between the repository object, or *parent*, and the application object, or *child*. If you want to change the object throughout your application, change the parent in the repository entry. The change is then inherited and propagated to the children of that parent repository object throughout your application.

Inheritance propagation happens automatically when you do all of the following:

- Name the parent objects (in the Name property under Identity in the Properties window); imported database objects are automatically named as a result of the import process.
- Copy the named parent widgets to your application screens, or identify the source of inheritance in your screen's and/or widget's `InheritFrom` property.
- Save the repository entry. Screens, service components, and widgets that inherit from the repository entry are updated appropriately.

You can also propagate inheritance by using the batch inherit utility `binherit` (page A-6).

What You Need to Know

The following items represent general information about repositories and inheritance as well as recommendations for using repositories to build applications:

- Although you can have multiple repositories, only one repository can be open at a time. Therefore, it is recommended that you create one repository per application.

- The `InheritFrom` property identifies the parent source of inheritance.
- Repository entries that are automatically created when you import database tables have the Inherit From property (under Identity) setting of `@DATABASE`.
- Repositories are a design-time tool. You don't need the repository to run your application.

Using the Repository

One of the first steps in your application development process should be to decide the role of the repository in your application. You need to decide what types of information will be stored in the repository, how that information will be used, and what properties need to be inherited.

Creating the Repository

Create a repository from options found in the editor. Since you can only have one repository open at a time, it is recommended that you create one repository per application. For the steps used to create a repository, refer to "How to Create a Repository" on page E-20 in *Using the Editors*.

In general, repositories are shared between all members of the development team. Therefore, you want to ensure that all users have access (and permissions) to read and/or write to the repository.

Opening a Default Repository

If the repository is named `data.dic` and is located in your application directory, Panther automatically opens it when you invoke the editor. You can also set the application variable `SMDICNAME` to open the repository of your choice.

Creating Repository Entries

Once the repository is created, you can populate it. A repository is like a Panther library, in that it is a collection of screens, service components, and reports that contain frequently used application objects. The screens, service components and reports that are placed in a repository are called repository entries.

There are several ways to create repository entries:

- Choose File→New→Repository Entry.
- Save any Panther screen, report or service component as a repository entry.
- Import your database tables, views or synonyms by choosing Tools→Import Database Objects. This creates a repository entry for each database table.

Creating Repository Objects

When you import database tables, the repository entries contain widgets corresponding to the database columns, and labels corresponding to the column names, but for other repository entries, you might want to create new objects. For example, you might have a repository screen containing push button templates for use throughout the application.

Since repository entries have complete access to all of the editor functionality, you can open a repository entry and make new objects from options on the Create menu. Alternatively, you can copy objects to the repository from any application component.

When you create objects on repository entries, make sure that the parent objects have a value in the Name property (under Identity). It is this name that is used in the Inherit From property in the child object to establish inheritance.

To view the repository entries in the current repository, choose View→Repository TOC.

Creating Screen Templates

If a series of application screens will share the same screen properties, a repository entry can be a screen template. For example, you might want a series of screens to share:

- Entry and/or exit functions
- JPL procedures
- Control strings
- Menu bars

Another use of screen templates in the repository would be to provide the definition for a screen that is used throughout the application, like an error screen or a dialog box.

Once an application component is created (screen, report, service component), it can inherit properties by setting the Inherit From property to the repository entry. The dialog box asks whether you want to inherit all properties. If you select Yes, all the properties for the application component are taken from the repository entry, overwriting any values you have set. If you select No, the Inherit From property is set to the repository entry but no property values are inherited. You can then set inheritance individually on a property-by-property basis by choosing the Inh button in the Properties window.

The widgets on the repository entry are not copied to the application component, just its properties.

Note: To define the colors for application screens, you might choose to define and edit settings in the configuration map file. Refer to “Configuration Map File” on page 45-25 for more information about `cmap` files.

Storing Database Information

Using the database importer in the editor (Tools→Import Database Objects), you can import a database table with all of its column definitions and primary and foreign key relationships into a Panther repository. If the database engine supports views or synonyms, those database objects can be imported as well.

When you import a database table to a repository, Panther creates a repository entry which includes a label and single line text widget for each column in the database table. The text widget has the column name as one of its properties in addition to other properties that are used for automatic SQL generation. A table view, available through the Widget List or the DB Interactions screen, is created which lists the columns in the database table and the primary key definitions for that database table. A link is created for each foreign key defined for that database table.

Once you have repository entries with your database information, you can copy those widgets to application components or use the wizards to build screens and reports for you.

One advantage of importing your database tables to the repository is that any changes to the database, such as the column length or column type, can be easily propagated throughout the application. The import database facility can be used to update the database repository entries. If the widgets on those screens are the parents of the widgets used in your application screens, changes in the column information are redefined for each child of that widget.

If you are planning to use the transaction manager, it is recommended that you copy the widgets corresponding to the database columns from database-derived repository entries. Repository entries created from the database import facility contain the necessary settings for SQL generation needed by the transaction manager.

Reimporting Your Database Tables

When you reimport your database tables to the repository, you can:

- Add new database columns to the repository entry for that table.
- Update the column length or column type for any database column.

However, if you delete database columns in your database, the widgets corresponding to those columns will *not* be deleted from the repository entry. You have the capability of adding widgets to a table view; the importer cannot distinguish between widgets that are former database columns and widgets that are table view additions.

Database Import Properties

The following properties are specified via the database import:

Column Edits

A subheading under Database provides access to additional properties that are acquired from the database column as opposed to Panther specifications for the widget.

- Length property—Specifies the column length (as a string) as defined in the database, if available.
- Scale property—Specifies the scale (as a string) of the column as defined in the database, if applicable. Scale determines the length of some numeric columns. This edit is currently unused by the SQL generator.
- Precision property—Specifies the precision (as a string) of the column as defined in the database, if applicable. Precision determines the number of decimal places in some numeric columns.
- Type property—Specifies the column type as defined in the database.

Storing Widget Templates

The repository can contain a master copy of any widget. For example, to use the same push button on several screens, you store a definition of that push button on a repository screen with the color of the push button, the pixmap and the control string that is invoked when that push button is selected. Then you can copy that push button to the applicable screens.

You can define widget templates on a repository entry, or you can copy a widget from one of your application screens to the repository. If you copy a named widget to the repository, the inheritance is automatically set for the widget on the application screen.

Storing Widget Definitions

If several screens share a set of widgets that work together throughout an application, you can create a repository entry for the widget set. In this case, either the widgets involved need property definitions that differ from the database repository entry or all of the widgets in the set are not located on the same repository entry.

First, import the database tables to the repository. Copy widgets from the database repository entry to a new repository entry. The widgets will inherit all the Database and Transaction properties. Set the properties that will not be inherited. Copy the widgets to application screens.

For example, in the Videobiz application, the `title_id` and `name` fields are often used together as a scrolling array. Instead of editing the array properties each time these widgets are used in an application screen, the widgets can be copied to a new repository entry, the necessary properties changed, and then the widgets can be copied to an application screen.

Using the Wizards

The first time you create a screen or report with the wizard, Panther copies the following entries into the open repository:

- `smwizard`—The template for the screen itself as well as the template for several objects found on the finished screen, including push buttons and grid widget.
- `smwizis`—The template for item selection screens. It contains push buttons and a grid widget; you can modify the properties for those objects or for the screen itself.
- `smwizsrv`—The template for service components using the JetNet/Tuxedo middleware adapter.
- `smwizrw`—The template for reports.
- `smwizweb`—The template for screens with Web-enabled output.

Since objects inherit from these templates, you can change the properties in the repository so that every new screen or report made with the wizard would inherit the desired settings.

If your repository is read-only, the wizard cannot make the necessary entries in the repository. Speak to your system administrator about adding these entries to the repository if you want to use the screen or report wizard.

Using Inheritance

For a screen or widget to inherit from a repository entry, the Inherit From property must be set to the designated object in the repository. This property is set automatically when objects are copied from the repository. For screens, the Inherit From property contains the name of the repository entry, for example, `msg_screen`. For widgets, the Inherit From property contains the name of the repository entry followed by the name of the object. For example, `titles!title_id` indicates that the widget inherits its properties from an object whose Name property is `title_id` on the repository entry `titles`.

For repository entries imported directly from the database, the Inherit From property for each widget is set to `@DATABASE`. When those widgets are copied to screens, the Inherit From property changes to the repository entry and object.

The repository object named in the Inherit From property is known as the parent. The widget containing the Inherit From value is known as the child. If a child inherits a property setting which is later changed in the parent, the change is propagated to the child when the screen is opened in the screen editor or after you update the screen using the `binherit` utility (page A-6).

An object in the repository can inherit from another repository object. An object on the screen can inherit from only one repository object.

Controlling Property Inheritance

To control the propagation of changes from parent objects to their children, use any of the following methods:

- Turn inheritance off for selected properties by selecting the property and toggling the Inh (Inherit) push button in the Properties window. Highlighted properties will still inherit changes.

  You can reinstate inheritance by selecting the property and choosing the Inh (Inherit) push button.

- Remove the Inherit From property value for the selected application object. Leaving a blank value removes the inheritance relationship for *all* properties; it does not remove the values. All future changes made to the repository entry are not propagated to the selected widget or screen.

- Choose Options→Inherit to temporarily turn inheritance off. When Inherit is active, the properties of each widget are updated and displayed in the editor as you work. When Inherit is inactive, the properties that are inherited are highlighted, but the values of those properties are not updated or displayed on the screen until inheritance is activated.

  This option is useful if you are in the process of changing or designing repository entries and don't want changes to propagate at design time. When you toggle the Inherit option back on, inheritance is restored and all changes are propagated appropriately to open screens and child widgets that inherit from the open repository.

Updating Inheritance in Application Screens

Use the `binherit` utility to update screens and reports from property values stored in the repository. `binherit` can also be used to report on the differences in the properties between the screens/reports and the repository.

Inheritance is updated each time you open a screen in the screen editor workspace and then save it. `binherit` performs this operation in batch mode, opening specified screens and saving them. Detailed information is located in the documentation for the `binherit` utility (page A-6).

Finding the Source of Inheritance

You can find a parent widget or screen (the one that inherits from) as well as find child widgets or screens (the ones that inherit).

The Edit→Find→Parent option opens the parent repository entry of the selected object. The Edit→Find→Children option finds the children of a repository object according to the specified criteria. For more information, refer to “Finding the Source of Inheritance” on page E-31 in *Using the Editors*.

CHAPTER

12 Creating Service Components

After creating the repository, distributed applications can create service components. These service components can be called by any client screen in your application.

All distributed application architectures in Panther use service components. However, the steps taken in the editor to define the service components and the coding required to call those service components can differ for each technology. For developers, this means that a service component created for one product can need modifications before it can work with another technology.

This chapter summarizes how a service component is created for different application architectures.

Service Components for JetNet and Oracle Tuxedo

In a Panther JetNet/Oracle Tuxedo application, a service can consist of three parts:

- A routine that implements the service.
- A service component (optional) that provides a physical means of sending, receiving, and processing data.

- A service definition in the JIF.

To create a service, you can: Use the screen editor or the screen wizard to create a service component—the graphical or visual representation of a service.

- Write the service routine that implements the service.
- Define the service in the JIF.

Creating Service Components in JetNet/Oracle Tuxedo

A service component is a graphical service. It looks, for the most part, like the client screen it is servicing. However, service components reside on the server (in a server library such as `server.lib`) and so are not visible to the user at runtime.

To create a service component using the editor, you must include and identify all the components necessary to implement the service. In addition to building the screen, you must code the service routines and set the appropriate property values on the client screens that will use the service component.

For the most part, service components must have the same contents and property values as the client screens that use them so that it can handle the data that flows between the client screen and the service. Therefore, changes you make on a client screen must also be made to its corresponding service component (if the changes are not implemented by a shared repository).

You can create new service components in the editor by choosing File→New→Service Component.

Alternatively, you can use the screen wizard to build service components as you build your client screens.

Writing Service Requests in JetNet/Oracle Tuxedo

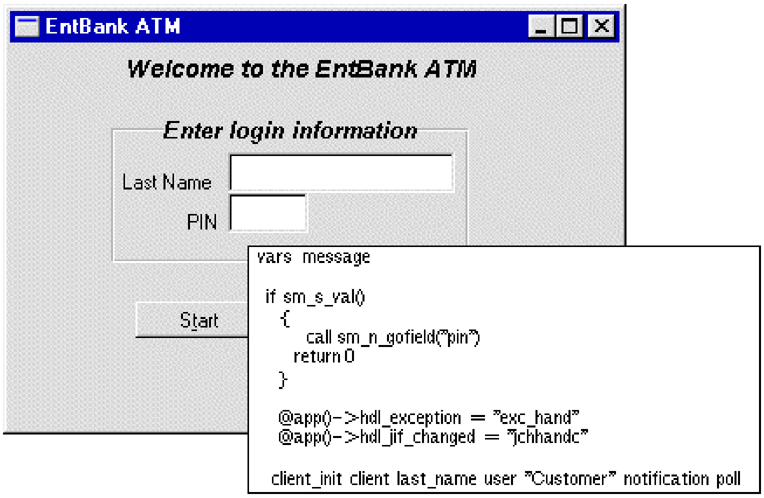
Service routines are responsible for receiving data from the client (if sent), performing some task, and returning data to the client (if requested).

Service routines are built for you when you use the screen wizard to create the service components. You can also write service routines the same way you write any other Panther application code. JPL is most convenient, but you can code a service as a C or Java function if that suits your application needs.

For information on writing service requests, refer to Chapter 12, “Creating Service Components,” in *JetNet/Oracle Tuxedo Guide*.

Creating Client Screens

When creating client screens, before any service requests are made, a screen must have a valid middleware connection. In order to open a middleware session for an application user, call `client_init`, as illustrated in the login screen from the EntBank sample ATM:



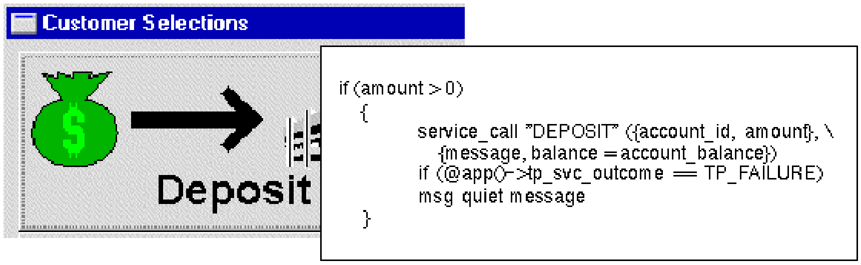
```
vars  message

 if sm_s_val()
   {
      call sm_n_gofield("pin")
    return 0
   }

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

  client_init client last_name user "Customer" notification poll
```

The `service_call` command initiates a service request, as illustrated in this excerpt from the deposit process:

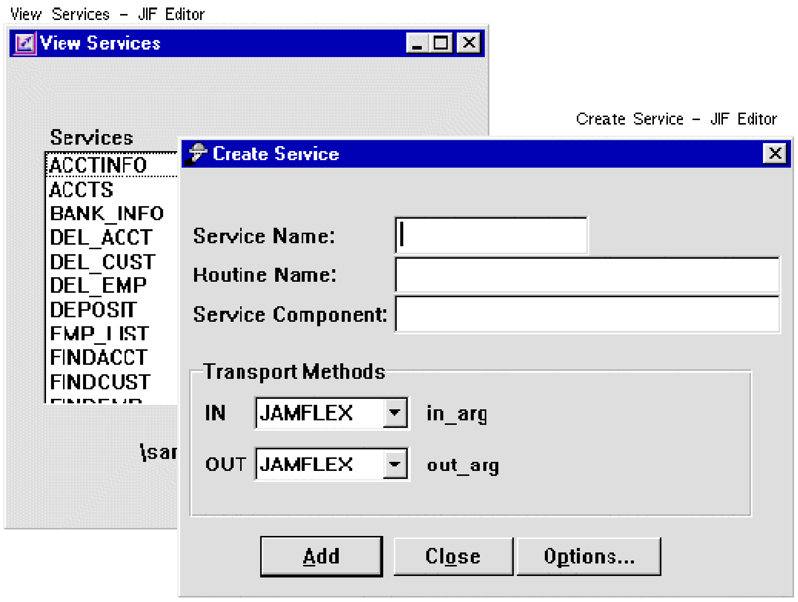


```
if (amount > 0)
 {
        service_call "DEPOSIT" ({account_id, amount}, \
          {message, balance = account_balance})
        if (@app()->tp_svc_outcome == TP_FAILURE)
        msg quiet message
 }
```

In order to close the middleware connection, call `client_exit`.

Updating the JIF

After you complete a service request, you must update the JIF to let the application know that the service is available. The JIF contains information about services and service groups and is used when a client makes a service call to determine the parameters, when a server needs to determine which C or JPL procedure should be executed to process the service call, and when a service forwards data to another service.



Service Name:
Routine Name:
Service Component:

Transport Methods
IN JAMFLEX in_arg
OUT JAMFLEX out_arg

Figure 12-1 The JIF editor defines the service: its service name, corresponding routine name, service component, and transport methods.

For information on using the JIF editor, refer to Chapter 24, “JIF Editor,” in *Using the Editors*.

Service Components for COM Components and EJBs

COM+ and Enterprise JavaBeans (EJBs) are object-oriented, component-based technologies. A COM component or an EJB has a public interface consisting of methods and properties that other objects or client applications can use to access the component. COM components can be deployed on any Windows system; EJBs can be deployed on IBM's WebSphere Application Server.

Creating Service Components in Component Applications

With Panther, you use a similar process to build service components deployed in these technologies. In fact, on Windows systems, saving your service component allows you to generate both the DLL needed for the COM component and the Java files needed for the EJB.

- In the editor, create a new service component by choosing File→New→Service Component.
- Define its properties and methods on the Component Interface window, available by choosing View→Component Interface.
- For COM components, define the application directory and other COM-specific settings on the COM section of the Component Interface window.
- For EJBs, define the target directory and other EJB-specific settings on the EJB section of the Component Interface window.
- Write the programming for each method using JPL, C/C++ or Java. Using the JPL verbs `receive_args` and `return_args` (or their corresponding C functions `sm_receive_args` and `sm_return_args`) you can pass data to and from the client screen.

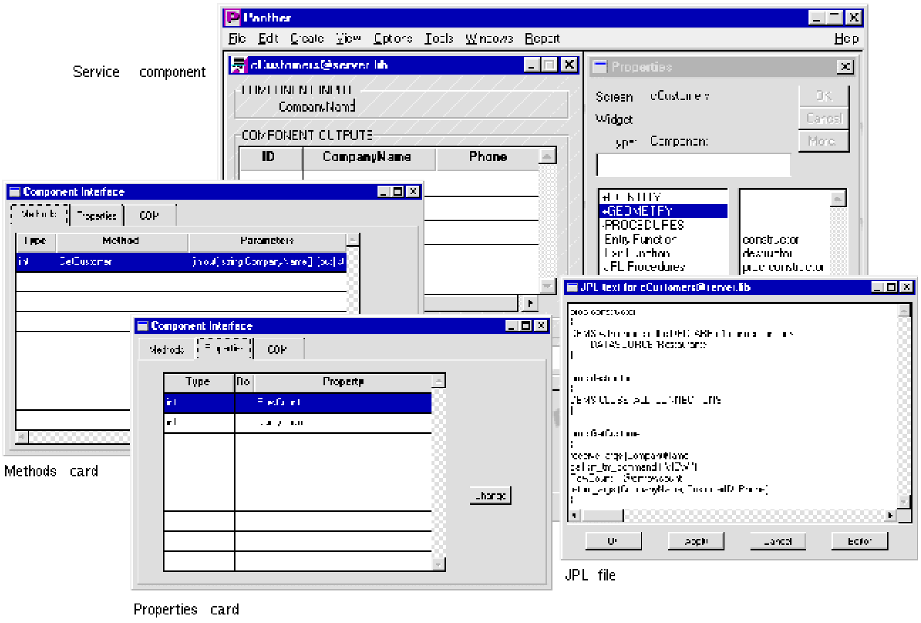


Figure 12-2 In the editor, you can create the service component, define its interface, and write the programming needed for its methods.

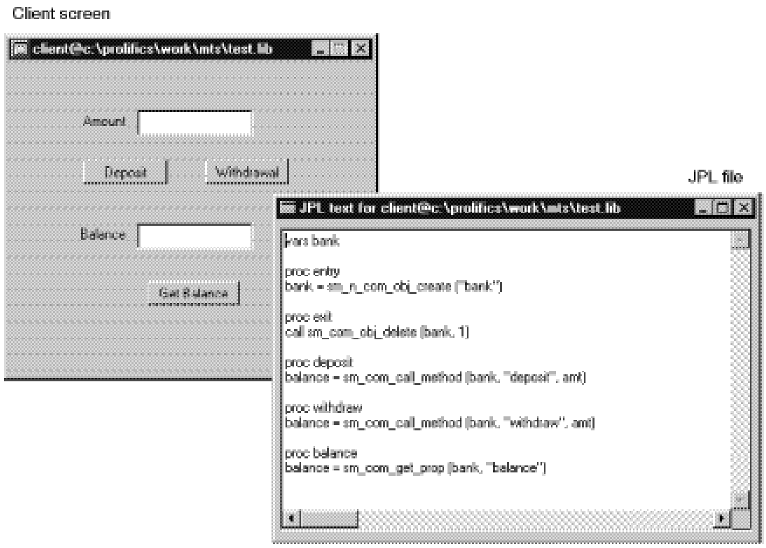
For more information about creating service components in the editor, refer to Chapter 7, “Defining Service Components,” in *Using the Editors* or to the following chapters: Chapter 3, “Building COM Components,” in *COM/MTS Guide* or Chapter 5, “Building Enterprise JavaBeans,” in *WebSphere Developer's Studio*.

Creating Client Screens in Component Applications

In the client screen, you first set the `current_component_system` property to specify the type of service components, `PV_SERVER_COM` for COM components or `PV_SERVER_EJB` for EJBs. You can then call the library functions which:

- Create the component, `sm_obj_create`.
- Call the component's methods, `sm_obj_call`.

- Get or set the component's properties, `sm_obj_get_property` and `sm_obj_set_property`.
- Destroy the component, `sm_obj_delete_id`.
- Control error messages, `sm_obj_onerror`, `sm_com_result` and `sm_com_result_msg`.


Client screen
client@c:\prolifics\work\mts\test.lib
Amount
Deposit
Withdrawal
Balance
Get Balance
JPL file
JPL text for client@c:\prolifics\work\mts\test.lib

```
vars bank

proc entry
bank = sm_n_com_obj_create ("bank")

proc exit
call sm_com_obj_delete (bank, 1)

proc deposit
balance = sm_com_call_method (bank, "deposit", amt)

proc withdraw
balance = sm_com_call_method (bank, "withdraw", amt)

proc balance
balance = sm_com_get_prop (bank, "balance")
```

Figure 12-3 In the client screen, you can instantiate a COM component or Enterprise JavaBean and access its methods and properties.

For more information about creating client screens which call COM components, refer to Chapter 4, "Building Client Screens," in *COM/MTS Guide*. For more information about creating client screens which call EJBs, refer to Chapter 7, "Building Client Screens," in *WebSphere Developer's Studio*.

Deploying Components in COM Applications

In a Panther COM/MTS application, a service component consists of:

- A service component, stored in a Panther application library
- A DLL file for the COM component

When you save the service component in a library, you are prompted to build the DLL for the COM component. Creating the DLL also creates the component's type library and the client registration file that is needed for DCOM deployment. Once built, COM components can be deployed under COM, DCOM, or MTS.

In a Panther COM application, both the Panther library containing the service component and the DLL file for the COM component must reside on the COM component server. The COM component must be installed on each server machine; each client must install either the COM component or its location in the registry.

For information about deploying service components in a COM+ environment, refer to Chapter 5, "Deploying COM Components," in *COM/MTS Guide*.

Deploying Components in WebSphere Application Server

For information about deploying service components and EJBs, refer to Chapter 6, "Deploying Enterprise JavaBeans in WebSphere," and Chapter 8, "Deploying Your Application," in *WebSphere Developer's Studio*.

Using the Common Component Interface

In order to provide a common programming interface for service components deployed under different technologies, the following C functions will work for both COM components and Enterprise JavaBeans:

- `sm_log`—Write a message to a server log.
- `sm_obj_call`—Call a service component's method.
- `sm_obj_create`—Instantiate the service component.
- `sm_obj_delete_id`—Destroy the component.
- `sm_obj_get_property`—Get the component's properties.
- `sm_obj_set_property`—Set the component's properties.
- `sm_obj_onerror`—Install an error handler.
- `sm_raise_exception`—Send an error code back to the client.

- `sm_receive_args`—Receive a method's parameters from the client.
- `sm_return_args`—Return a list of parameters back to the client.

You must first specify the current_component_system property to determine the type of components currently in use: `PV_SERVER_COM` for COM components or `PV_SERVER_EJB` for EJBs deployed in WebSphere Application Server.

CHAPTER

13 Developing Client Screens

The client side of a Panther application is largely composed of screens: client screens that users open as forms or windows and screens that are saved as repository entries or that are used to merge data into other screens or widgets as LDBs.

In two-tier applications, client screens contain both the application logic and the presentation interface. In three-tier applications, client screens are primarily concerned with the presentation interface and send service requests to the application server for processing.

This chapter discusses:

- How to open and close screens
- How a screen can be opened as a dialog box
- The size and position of screens
- The scope of screen processing
- Screens' runtime properties

Other chapters in this manual discuss:

- How to move data between screens (Chapter 25)
- How the stack of screens is maintained (Chapter 24)
- How to display messages in a window, on the status line, or as a dialog (Chapter 26)

Creating Screens

You can build screens with the screen wizard or from scratch in the editor. Once the screen is created, save it in the appropriate library.

For information on creating screens and saving them in libraries, refer to Chapter 6, “Defining Screen Properties,” in *Using the Editors*. For instructions on using the screen wizard, refer to Chapter 4, “Screen Wizard,” in *Using the Editors*.

Creating Dialog Boxes

A screen that has its Dialog (dialog) property set to Yes:

- Cannot be resized, maximized, or minimized at runtime. These options are not available on the screen's system menu, and the border (specifically in GUIs) has no maximize/minimize buttons. However, it can be moved.
- Is modal at runtime, the user is forced to enter data, respond to, or acknowledge the dialog box before proceeding with the application. The menu bar and controls strings outside of the dialog box cannot be accessed while the modal dialog box is active.

Any stacked or sibling windows that are invoked from a dialog box are also opened as modal dialog boxes.

To prevent users from closing a dialog box from the system menu, set the screen's Close Item (`close_item`) property to No.

For an example of a dialog box using tab widgets, refer to “Creating a Tab Dialog Screen” on page 16-5 in *Using the Editors.*

Understanding Screen Scope

A screen provides a namespace for widgets, JPL procedures and Panther variables.

For widgets, this means that all widgets on a screen must be uniquely named, even though a widget with the same name can appear on other screens.

For variables and JPL procedures, this means that Panther searches the current screen for the variable or procedure before going to public modules.

A screen is also one of the levels for menu scope. Setting the screen's Menu Name (`menu_name`) property overrides the application-level menu.

Opening Screens

Applications typically let users open screens by pressing a key, choosing a menu item, or a selection-type widget or push button. You specify the screen to open through the control string property of the screen, menu item, or widget; the control string specifies which screen to display and whether to open it as a form or window. For example:

This control string:	**Opens the screen as a:**
`screen-name`	Form
`&screen-name`	Stacked window
`&&screen-name`	Sibling window

For information about the form stack and the window stack, refer to Chapter 24, "Setting the Screen Sequence."

You can also use Panther runtime functions to open a screen and give it focus:

- `sm_jform` opens the specified screen as a form. It first closes all open screens—that is, the previous top form and any windows in the window stack.
- `sm_jwindow` and `sm_r_window` open the specified screen as a window.

Notes: Avoid calling `sm_jform` in a screen entry or exit function. Doing so can yield unpredictable results. To open a form at screen entry, use the built-in function `jm_keys`; pass as its argument a function key with a control string that brings up the desired window. You can call `sm_jwindow` and `sm_r_window` in screen entry and exit functions if you close the window before the function returns.

For information about screen entry events, refer to page 17-4, "Screen Entry."

Search Path

Panther looks for the named screen in the following places in this order:

- The memory-resident screen list.
- All open libraries.

If all searches fail, Panther displays an error message and returns.

Screen Display Defaults

Unless otherwise specified, Panther tries to display the entire screen. If a screen is opened as a form, Panther displays it at the physical display's upper-left corner; in GUIs, this excludes the menu bar, which remains visible. If a screen is opened as a window, Panther tries to leave the calling screen's last cursor position visible. In GUI environments, the displayed form always leaves the menu bar visible.

Displaying Screens in Viewports

Panther automatically handles screens whose size exceeds the actual dimensions of the viewing area—for example, the screen is larger than the physical display. When a screen's dimensions exceed its display area, Panther displays the screen in a viewport with vertical and horizontal scroll bars, so users can scroll out-of-view data into view. By default, the viewport's upper-left corner (1,1) initially displays the screen's upper-left contents, unless this prevents display of the cursor. Panther always ensures that the cursor's initial position in a viewport—usually the first field—is visible. If necessary, it adjusts the screen offset within the viewport accordingly.

The viewport itself can only be as large as the system's physical or virtual display. In character mode, the two are identical; thus, a viewport can only be as large as the screen. In contrast, under some GUIs—for example, Motif—a viewport can be larger than the physical display. The offscreen portions of the viewport can be brought into view either by the user or programmatically.

Overriding Display Defaults

The control string that you use to open a screen can specify the screen's position and dimensions. If the specified dimensions are unable to display the entire screen, you can also specify the offset of the screen within the viewport. The full control string syntax is as follows:

```
[lead-char] (row, col, height, width, vrow, vcol) screen-name
```

If you omit `lead-char`, Panther opens the screen as a form. A single ampersand (&) opens the screen as a stacked window, while a double ampersand (&&) opens it as a sibling window. If you use ampersands (& or &&) to open a screen as a window, they must precede the viewport arguments. Parentheses must enclose all viewport arguments.

For example, the following control string specifies the PF1 key to open the `new_customer` screen at the upper left corner of the physical display:

```
PF1 = (1,1)new_customer
```

For more information, refer to Chapter 18, “Programming Control Strings.”

Specifying Viewports at Runtime

The runtime functions `sm_jform`, `sm_r_window`, and `sm_jwindow` can specify viewport parameters. For example, the following calls to `sm_jwindow` and `sm_r_window` are equivalent: each opens `myscreen` as a stacked window at coordinates 4,4 on the physical display:

```
ret = sm_jwindow("&(4,4)myscreen");
ret = sm_r_window("myscreen", 4, 4 );
```

Opening Screens in Windows Applications

In a Windows application, screens are implemented inside an MDI (multiple document interface) frame. MDI allows several windows to be opened inside a main application window.

Your Windows initialization file contains several settings controlling the use of the MDI frame. For more information, refer to Chapter 3, “Windows Initialization File,” in *Configuration Guide*.

The following library functions deal specifically with MDI frames:

- `sm_mw_get_client_wnd` gets a handle to the client section of an MDI frame.
- `sm_mw_get_frame_wnd` gets a handle to the MDI frame of an application.
- `sm_mw_PrintScreen` sends either the current Panther screen or all the screens in the MDI frame to the printer.

Specifying the Window Style

As of Panther 4.5, there are new options for opening windows under the Windows operating system. The new options are implemented by the properties Keep in Frame and Topmost. These are screen-level properties and are found under the Identity section in the Properties window.

Previously, windows could either be opened as MDI windows or as dialogs. An MDI window cannot be moved outside the MDI frame. A dialog can be moved outside the MDI frame, but blocks access to the MDI frame – when a dialog is opened, focus cannot be given to any MDI windows, nor can the MDI menu bar be accessed until the dialog is closed.

With the new options, a screen can open as a non-MDI window, moving outside the MDI frame, but not blocking access to the MDI windows or to the MDI menu bar. You can also specify this screen to be the topmost window.

For more information, refer to “Specifying Styles under Windows” on page 6-25 in *Using the Editors*.

Closing Screens

You can close a screen with `jm_exit` and `sm_jclose`. The two functions are equivalent; `jm_exit` is a built-in function, while `sm_jclose` is an installed library function.

By default, the `EXIT` logical key calls `jm_exit` and causes the current screen, whether a window or form, to close. If you leave `EXIT` unassociated with any control string, you can make it available to users to exit the current screen—for example, by pressing its physical key (Esc on most terminals) or by attaching it to a menu item or push button.

For information about screen exit events, refer to page 17-19, “Screen Exit.”

Setting Screen Properties

By setting properties in the Properties window, you can define the appearance of the screen in your application. For more information, refer to Chapter 6, “Defining Screen Properties,” in *Using the Editors*.

When you create a screen, Panther initializes its properties according to internally set defaults. You can set a screen to inherit properties from a repository entry through the screen's Inherit From property. When you do this, Panther writes the entry's properties to the target screen. You can subsequently turn inheritance on and off for individual properties, or turn off inheritance for the entire screen by emptying its Inherit From property.

Some screen properties listed in the editor are accessible at runtime. In addition, there are runtime-only properties for screens. There are also runtime properties for screens; For a list of all runtime screen properties, refer to “Screen and Frameset Properties” on page 1-107 in *Quick Reference*.

Using JPL to Set Screen Properties

You can get and set all screen properties at runtime through JPL, which contains two screen objects:

`@screen`	The name of a Panther screen that is on the window stack. To specify the active window, supply `@current` as a string.
`@screen_num`	The number of a Panther screen that is on the window stack, where 0 is the active window, -1 is the window below it, and so on. Positive numbers number from the bottom of the window stack: 1 is the base window, 2 refer to the window above it, and so on.

For example, this statement gets the title property for screen vidlist.scr:

```
cur_title = @screen("vidlist.scr")->title
```

The following example sets the title property:

```
@screen("vidlist.scr")->title = "Current Title List"
```

For more information about accessing properties at runtime, refer to page 19-33, “Setting Properties Using the Property API.”

Runtime Properties for Screens

In addition to the properties listed for screens in the Properties window, there are screen properties that are only available at runtime. The screen will also be affected by the application's runtime properties.

`numflds`
: Returns the number of fields on the current screen.

`numgrps`
: Returns the number of groups on the current screen.

`sibling`

Set the screen as a sibling window. Refer to “Sibling Windows” on page 24-4 for the discussion of sibling windows.

CHAPTER

14 Identifying Screen Widgets

Manipulating widgets at runtime requires that you be able to uniquely identify each widget. In order to identify widgets, it is recommended that you name each widget.

This chapter describes how to identify each widget on the screen, how to identify each occurrence of a widget, and how to determine the contents of a group and of an ActiveX control.

Functions described in this section are documented in the *Programming Guide*; refer to that manual for the syntax and specific behavior of each function.

Widget Types

You can create widgets in the editor by choosing the widget type on the Create menu or on the Create toolbar. Table 14-1 lists the widget types, their availability and their description in the *Using the Editors*; that manual also contains instructions for setting property values in the Properties window.

Table 14-1 Widget types and their platform availability

Widget type	Motif	Windows	Web	Char mode
ActiveX controls		●	●	
Boxes	●	●	●	●
Check boxes	●	●	●	●
Combo boxes	●	●		●
Dynamic labels	●	●	●	●
Graphs	●	●	●	
Grids	●	●	●	●
Lines	●	●	●	●
Links	●	●	●	●
List boxes	●	●	●	●
Multiline text	●	●	●	●
Option menus	●	●	●	●
Push buttons	●	●	●	●
Radio buttons	●	●	●	●
Scales	●	●		●
Single line text	●	●	●	●
Static labels	●	●	●	●
Tab controls	●	●		
Table views	●	●	●	●
Toggle buttons	●	●		●

Widget Identifiers

JPL and most runtime functions let you identify widgets by object ID, name, or number. Panther provides three widget identity properties, accessible through JPL or the library function `sm_prop_get`. All three properties are read-only. In JPL, these properties are:

- `id`—Set to the integer identifier that Panther assigns to each widget at runtime.
- `name`—Set to the name that the developer assigns to this widget in the screen editor.
- `fldnum`—Set to the widget's base field number.

In JPL, you can identify the current widget (the widget that has focus) with statements that use the `@current` object identifier as follows:

```
@widget("@current")->widgetProperty
```

For example, the following statement sets variable `cur_widget` to the current widget's id property:

```
cur_widget = @widget("@current")->id
```

You can also get a widget's id property by calling `sm_prop_id`, as in this statement:

```
cur_widget = sm_prop_id(@widget("@current"))
```

For more information on referencing widgets in JPL, refer to page 19-33, “Setting Properties Using the Property API.”

Object IDs

At runtime, Panther assigns each widget a unique object ID, which provides the most reliable way to reference and manipulate that widget. Widget and screen IDs are set when a screen initially opens and remain valid as long as the screen remains on the window stack. All object ID assignments during an application's life span are unique; IDs that are no longer valid are not reused.

You can get a widget's ID through its id property or by calling `sm_prop_id`.

Widget Names

Each widget can be assigned a name through its Name property in the screen editor. Widgets that are created as the result of importing database tables are automatically named—corresponding to the database column. A widget name can be up to 31 characters and can start with an alphabetic character, an underscore, a dollar sign, or a period. Names are case-sensitive; thus, city and City are two distinct names.

Names must be unique within a screen. Widgets on different screens can share the same name, but they should use the same name only when they share data through an LDB entry or inherit the same properties.

A widget must be named when one of these conditions is true:

- Its contents are shared with other screens through a local data block (LDB).
- It inherits its properties from a repository entry of the same name.

For more information about mapping database columns to widgets, refer to Chapter 29, "Reading Information from the Database."

Field Numbers

Widgets that allow data entry all have internally assigned field numbers. Field numbers are assigned automatically when you add a widget to a screen, and are reassigned whenever the widget position changes. Panther numbers widgets as follows:

- Widgets are numbered sequentially from left to right, and top to bottom—as Field #1, Field #2, and so on.
- Each onscreen occurrence, or element, of an array has a unique field number. The number of the first element in an array, or the array's base field, is the number by which the widget as a whole is identified.

Notes: Elements in an array might not be numbered contiguously, depending on whether other widgets are positioned on the array's right side. Refer to the next section on arrays for more information about element numbering.

If you rely upon field numbers, be aware of two potential drawbacks:

- Because a widget's position determines its number, changing design considerations and runtime repositioning make referencing widgets by number problematic.
- Widgets that do not contain data, such as static labels, boxes, lines, and grid widgets, are not numbered and so have no `fldnum` property that you can use to reference them.

In general, names and object IDs are more reliable handles for identifying widgets and controlling their behavior at runtime; named widgets are also easier to identify in your code.

Arrays

Panther identifies an array as any widget that can contain one or more occurrences of data. In this sense, any Panther widget type that can contain data can be regarded as an array. Typically, however, there are three widget types regarded as arrays: single line text, multiline text, and list box. In all three cases, you can modify the Geometry properties of these widgets to allow more occurrences than are visible onscreen. Widgets thus defined are scrolling arrays.

An array consists of elements and occurrences:

- The number of elements in an array is determined either by its `array_size` property or, if within a grid widget, by the grid widget's `onscreen_rows` property. All elements in an array are visible whether or not they contain data.
- Occurrences are the data that populate an array, visible via the array elements. Occurrences are numbered independently of elements, between 1 and the number of occurrences that currently populate the array—obtained through the runtime property `num_occurrences`.

Panther allocates memory only for occurrences that have data; trailing occurrences that are empty are discarded. Information is maintained about the number of occurrences allocated for an array and the offset of occurrences within an array's elements.

Non-Scrolling and Scrolling Arrays

If an array is non-scrolling—its scrolling property is set to `PV_NO`—it can only have as many occurrences as elements. For example, if a non-scrolling array's `array_size` property is set to `3`, the array can contain a maximum of three occurrences of data.

In a scrolling array— scrolling is set to `PV_YES`—occurrences can outnumber elements if its `max_occurrences` property is greater than its `array_size` property. A scrolling array can contain up to `max_occurrences` occurrences; if this property is set to `NULL`, the array can contain an unlimited number of occurrences.

Synchronized Scrolling Arrays

Scrolling arrays can be synchronized so that they scroll together. This helps manage related information in table-like screens.

Panther automatically synchronizes arrays when they meet either of the following conditions:

- The widgets are grid members within a grid widget.
- They are database-derived widgets that belong to the same table view or to different table views that are joined by a server link.

You can manually synchronize arrays that do not meet the above conditions by making the widgets members of a synchronized scrolling group.

Refer to “Synchronizing Scrolling Arrays” on page 8-20 in *Using the Editors* for instructions on creating synchronized arrays.

Element and Occurrence Numbering

At runtime, Panther numbers all occurrences between 1and n, where n has the value of the array's runtime property `num_occurrences`. In a non-scrolling array, the first occurrence—referenced in JPL as array-spec[1]—is always visible in the array's first element—in JPL, array-spec[[1]]; the second occurrence is in the second element; and so on.

In a scrolling array, the first occurrence and first element coincide when the first occurrence is visible in the first element; if the first occurrence scrolls out of view, the occurrence that is visible in the array's first element can be any number up to and including array-spec->`num_occurrences`. For example, Figure 14-1 shows scrolling array `pid` in which the first occurrence—`pid`[1]—is visible in the array's first element—`pid`[[1]]:

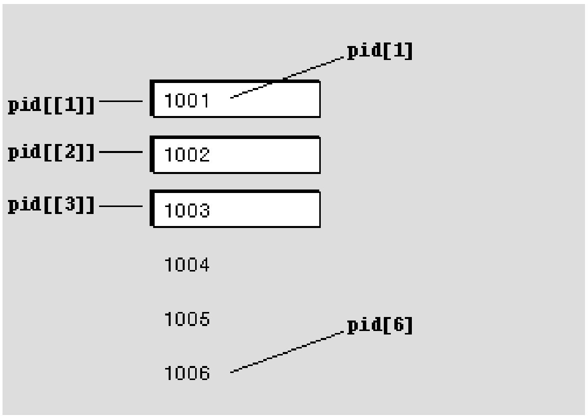


Figure 14-1 The first occurrence is displayed in the array's first element.

Figure 14-2 shows the same array; however, the data has scrolled up one occurrence so the first occurrence `pid`[1] is out of view; the first element `pid`[[1]] now contains the second occurrence `pid`[2]:

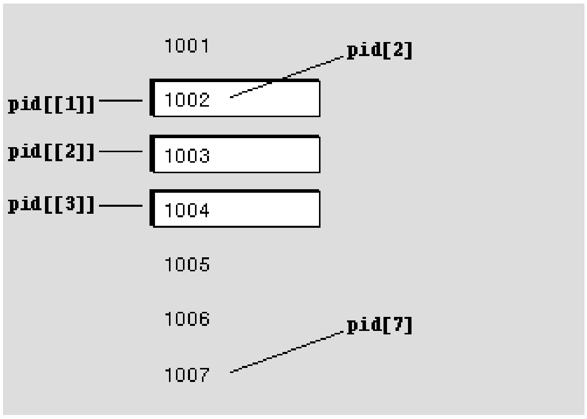


Figure 14-2 The first occurrence is scrolled offscreen; the second occurrence is therefore displayed in the array's first element.

You can obtain the current occurrence in an array's first element through the array's `first_occurrence` property. For example, given array pid's state in the first figure, its `first_occurrence` property is set to 1; in the second figure, `first_occurrence` is set to 2. You can also use this property to programmatically scroll an array's occurrences. For example, the following JPL statement resets a scrolling array so that its first element displays the first occurrence of data:

```
pid->first_occurrence=1
```

If an array is referenced in a JPL procedure without an occurrence being specified, Panther uses the default occurrence. When executing a field entry, field exit, or validation function, the default occurrence is the occurrence currently being processed. Otherwise, the default occurrence is 1.

Groups

You can group widgets of the same or different types together. This allows you to perform such tasks as allowing synchronized scrolling among several arrays or allowing selection among radio buttons or check boxes. Widgets within each group retain their separate identities; however, Panther also recognizes groups as unique components that can be named, and identifies their constituent widgets by their relative offset within the group. All group properties are accessible at runtime through JPL and by Panther library functions `sm_prop_get` and `sm_prop_set`.

If a widget is a member of a group, the following runtime properties return the group's object ID:

- `group`— the widget is a member of a selection group.
- `sync_group`—the widget is a member of a synchronized scrolling group.

If a widget is not a member of that type of group, these properties return an empty string.

Two library functions let you identify groups and their widgets:

- `sm_i_gtof` converts a group name and group occurrence into a field number and occurrence. This function lets you use other Panther library functions to manipulate group widgets by converting group references into widget references. For example, to access text from a specific widget within a group, use `sm_i_gtof` to get the field and occurrence number, then call `sm_o_getfield` to retrieve the text.
- `sm_ftog` converts field references to group references. It returns the name of the group that contains the referenced widget and the widget's offset within the group.

For information on traversing members of a group, refer to page 23-3, "Traversing Widgets."

Table views are also considered group widgets. Refer to "Identifying a Widget's Table View" on page 36-24 for information on identifying the table view at runtime.

ActiveX Controls

Active X controls, available for Windows and Web applications, are considered separately since the ActiveX control itself is not a Panther' widget, only the ActiveX container is. The Active X container's CLSID property (`clsid`) determines which ActiveX control is located inside the container. If the ActiveX control specified in that `CLSID` property is registered on your system, the control will be displayed in the editor and the Properties window's ActiveX category will display the control's property names and settings.

For more information on ActiveX controls, refer to Chapter 18, "ActiveX Controls," in *Using the Editors.*

CHAPTER

15 Including Menus and Toolbars

Menu bars, popup menus, and toolbars are all instantiated from menus that you define through the menu bar editor and save to a binary resource file, or menu script. Because menu bars, popup menus, and toolbars are created from the same menu definition, runtime access to all three is provided through the same set of library functions. In this chapter, all references to menus apply equally to menu bars, popup menus, and toolbars, unless otherwise noted.

Menu definitions are saved in menu scripts. When you save a menu through the menu bar editor, the menu and its submenus are saved to a binary script. At runtime, Panther can load one or more scripts into memory; it can then install menus from these scripts at different levels of the application. Depending on how a menu is installed, it can display as a menu bar on a screen or be invoked as a popup from a screen or widget. If the menu is installed as a menu bar and one or more of its items have their `MNI_DISPLAY_ON` property set to `DISPLAY_TOOL` or `DISPLAY_BOTH`, Panther also displays a toolbar with the menu bar.

You can specify to load a menu script and install a menu from the Properties window of a screen or widget. Alternatively, you can use Panther runtime functions to load and display menus.

This chapter shows how to perform the following tasks:

- Load menus into memory.
- Install menus for display with a screen or widget.
- Display menu items on a toolbar.
- Change menu properties at runtime.

- Remove menus from display and unload them from memory.
- Use `m2asc` to convert menus from binary to ASCII format, and vice versa.
- Read menu definitions in ASCII format.

Loading Menus into Memory

When you load a menu script, all of its menus are stored in memory and are available for installation and display. Panther applications have three levels of memory for loading menus:

- *Application memory.* Menus that are loaded into application memory are accessible throughout the application.
- *Screen memory.* Each screen has its own memory; menus that are loaded into a screen's memory are available only to that screen and its widgets.
- *Field memory.* Most widget types have their own memory; menus that are loaded into a widget's field memory are available only to that widget.

A script can be loaded only once in each memory location—that is, a given script can be loaded only once into application memory, and once into the memory location of a screen or widget. So, if several screens have the same menu installed from a script in application memory, they display identical menus—if one menu changes, those changes are written to the same memory and immediately propagated to the other menus. Alternatively, if each screen has the same menu installed from its own memory—each screen has its own instance of the script loaded into screen memory—each instance of that menu is unique: changes to one are written only to its own memory and have no effect on the other screen menus. This chapter contains sections on “Installing Menus with Shared Content” on page 15-5 and “Installing Menus with Unique Content” on page 15-6.

You can load a menu script in two ways:

- Enter its name in the screen's Menu Script File property or a widget's Popup Script File property.

- Call the library function `sm_mnscript_load`.

The first method loads the menu script into the screen or widget's memory and makes its menus available to that screen or widget. `sm_mnscript_load` can load the specified script into any memory location that is the same or higher than its caller, as shown in the following table:

sm_mnscript_load caller	**Valid memory locations**
Application	Application
Screen	Current screen Application
Widget	Current widget Current screen Application

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 its own memory.

Installing Menus

After you load a menu script, you can install any of its menus for display. When a menu is installed, Panther finds it in the specified script and reads its definition. If the menu contains external references—the menu is defined in another script—Panther resolves these; it then makes the menu available for display.

Except for Motif versions, Panther applications can display only one menu bar and its corresponding toolbar at a time. For example, if an application contains multiple screens and each screen has its own menu, only the menu bar and toolbar of the active screen are displayed. Under Motif, an application menu and a screen menu can display simultaneously if you set the `baseWindow` and `formMenus` resources to `true`.

You can install a menu at four scopes:

- Application scope. A menu that is installed at application scope displays with all screens unless the active screen has its own menu. Under Motif, the application menu displays with the base window if the `baseWindow` resource is set to `true`. You can install an application menu only from application memory.
- Screen scope. A menu that is installed with a screen displays whenever its screen is opened or reexposed. This menu is also used by successive screens that lack their own menu. You can install a screen menu from application or screen memory.
- Screen popup scope. A menu that is installed as a screen's popup can be invoked by the user 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 application or screen memory.
- Field scope. A menu that is installed with a widget displays as a popup that the user invokes when that widget has focus. You can install a menu for a widget from any level of memory—application, screen, or field.

You can install a menu in two ways:

- Enter its name in the screen's Menu Name property or in the widget's Popup Menu property. You can also enter a menu name in the screen's Popup Menu property.
- Call the library function `sm_menu_install`. You must use this function to install menus at application scope.

When a screen opens, Panther looks at its Menu Name property and installs the menu specified there, if any, as that screen's menu bar. If any of the menu items have their `Toolbar` property set to `Yes`, Panther creates a toolbar from the images associated with those items and displays it below the menu bar.

If the screen's Popup Menu property specifies a menu, Panther also installs this menu at screen scope. Panther displays the screen's popup menu when the user invokes it from the screen. If no entry exists for Popup Menu, Panther also uses the Menu Name property for the screen's popup menu.

At screen open, Panther also checks the Menu Popup property of each widget; Panther installs each menu specified by a widget at field scope and displays it as a popup when invoked from that widget.

With sm_menu_install, you can install a menu at any scope that is the same or higher than the calling environment, from any memory location that is valid for that scope. Thus, a screen's entry procedure can install a menu for the current screen or for the application, while a widget's entry procedure can install a menu for the current widget, its screen, or the application. If another menu is already installed at the specified scope, it is removed. 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. 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 widgets; if you do, all instances of this menu are always the same. If you install the same menu for different widgets from screen memory, all popup menus of those widgets are identical.

For example, the following entry 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:

```
proc install_menu
if (sm_mnscript_load(MNL_APPLIC, "mnscript_myprog") == 0)
   {
      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 the application can install their own instances of this menu with this call:

```
call sm_menu_install \
   (MNS_SCREEN, MNL_APPLIC, "mnscript_myprog", "scr_mn")
```

All screens that display `scr_mn` as a menu bar and toolbar display the same menu and toolbar. Thus, if one screen makes a menu item inactive, that item is inactive on the other screens.

Installing Menus with Unique Content

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—specify the script in the screen's Menu Script File property; or call `sm_mnscript_load` at screen entry with an argument of `MNL_SCREEN`.
2. Install the menu from screen memory—specify the menu in the screen's Menu Name property or Popup Menu property; or call `sm_menu_install` at screen entry with arguments of `MNS_SCREEN` (for a menu bar) or `MNS_SCRN_POPUP` (for a popup menu), and `MNL_SCREEN`.

Similarly, you can make sure that widgets have unique copies of the same popup menu. Repeat these steps for each widget:

1. Load the menu script into field memory for the widget—specify the script in the widget's Menu Script File property; or call `sm_mnscript_load` at widget entry with an argument of `MNL_FIELD`.
2. Install the menu from the widget's memory—specify the menu in the widget's Menu Name property; or call `sm_menu_install` at widget entry with arguments of `MNS_FIELD` and `MNL_FIELD`.

Referencing 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 either memory location, Panther displays an empty submenu.

Displaying Toolbars

A screen can display a toolbar alongside or in place of a menu bar. Both the toolbar and menu bar are instantiations of the same menu: any item that can be displayed on the screen's menu bar can also be displayed on its toolbar, and vice versa.

Display of a toolbar depends on two conditions being true:

- Toolbar display is enabled.
- At least one screen menu item is set for toolbar display.

You enable toolbar display through the setup variable `TOOLBAR_DISPLAY`, which can be set to `TOOLBAR_ON` (the default) or `TOOLBAR_OFF`. This variable can be changed at runtime by calling `sm_option` to toggle toolbar display for the entire application.

Display of individual items on a screen's menu bar and/or toolbar is determined by their `MNI_DISPLAY_ON` property, which is set to one of these values:

- `DISPLAY_MENU`: Display the item only on the screen's menu bar (default).
- `DISPLAY_TOOL`: Display the item only on the toolbar.
- `DISPLAY_BOTH`: Display the item on both the menu bar and toolbar.
- `DISPLAY_NEITHER`: Suppress display on menu bar and tool bar.

You can set a menu item's initial display in the menu bar editor and change it at runtime.

Panther for Windows can dock the current toolbar to the MDI frame. Runtime application properties control the position and appearance of the toolbar; refer to page 15-10, “Dockable Toolbar Properties.”

If a menu item is set to display on a toolbar, you should set its pixmap properties to determine the item's display in its different states. You must also set pixmap properties for all toolbar items.

Panther for Windows uses MFC to control toolbar display, which sets the item's inactive and armed display from the `MNI_ACT_PIXMAP` (active) property and the display for MouseOver events from the `MNI_HOT_PIXMAP` (hot) property.

Pixmap property		**Platform availability**
`MNI_ACT_PIXMAP`	active	Motif, Windows
`MNI_INACT_PIXMAP`	inactive	Motif
`MNI_ARM_PIXMAP`	armed	Motif
`MNI_HOT_PIXMAP`	hot	Windows

For more information about setting pixmap properties, refer to “Displaying Pictures on Toolbar Items” on page 25-14 in *Using the Editors*.

You can set an item's tooltip text, which displays when the cursor remains above that item; tooltip display is enabled or disabled for the entire application through the setup variable `TOOLTIP_DISPLAY`. You can toggle tooltip display on and off by using `sm_option` to set it to `TOOLTIP_ON` (default) and `TOOLTIP_OFF`, respectively.

You can control the font type and size for tooltips in Motif applications through the Panther resource file. For example, this statement sets tooltip text to 18 point Helvetica:

```
Panther*toolbar*tooltip.fontList:      *-helvetica-*-18-*
```

On Windows, the appearance of tooltip text is under MFC control.

Changing Menus at Runtime

Panther provides a set of library functions that let you change menus and their items at runtime. You can:

- Get and set menu and menu item properties.
- Change the state of toggle items.
- Create and delete menus and menu items from memory.

Getting and Setting Properties

All properties that are available through the menu bar editor also are accessible and modifiable through Panther library functions.

You can get the current setting of a menu property by calling either `sm_menu_get_int` or `sm_menu_get_str`. To get a menu item's property setting, call either `sm_mnitem_get_int` or `sm_mnitem_get_str`. Use the `_int` variant for those properties that have an integer value—for example, `MN_TEAR` or `MNI_ACTIVE`; use the `_str` variant for properties that take string values, such as `MN_TITLE` and `MNI_CONTROL`.

`sm_menu_bar_error` lets you test error conditions generated by the aforementioned `_get` functions. These functions return the value of the specified property when successful; otherwise, they return -1 for failure of the `_get_int` variants and `NULL` for the `_get_str` variants. `sm_menu_bar_error` returns the error code generated by the last call to one of these variants.

`sm_menu_change` and `sm_mnitem_change` set menu and menu item properties, respectively. These properties are derived from a memory-resident script. Because these functions change the specified script, all instances of menus installed from this script get the requested property change.`sm_mnitem_change` and its variant `sm_n_mnitem_change` cannot be called directly from JPL; consequently, a number of wrapper functions are declared and installed, which you can use to modify menu items in JPL modules.

Dockable Toolbar Properties

For Panther for Windows, the following runtime application properties control the position and appearance of the toolbar in relation to the MDI frame:

`PR_TOOLBAR_ALLOWED_SITES`
: Set the frame placements allowed for the toolbar using one or more of the following bit flags:

`PV_TOOLBAR_FLOAT`
`PV_TOOLBAR_TOP`
`PV_TOOLBAR_BOTTOM`
`PV_TOOLBAR_LEFT`
`PV_TOOLBAR_RIGHT`

`PR_TOOLBAR_CURRENT_SITE`
: Set the current placement of the toolbar using one of the defined bit flags: `PV_TOOLBAR_FLOAT`, `PV_TOOLBAR_TOP` (default), `PV_TOOLBAR_BOTTOM`, `PV_TOOLBAR_LEFT`, or `PV_TOOLBAR_RIGHT`.

`PR_TOOLBAR_HIDDEN`
: Set whether the toolbar is currently displayed using `PV_YES` and `PV_NO`. Users can hide the toolbar by clicking on the X in the upper-right corner of the menu.

`PR_TOOLBAR_X_POSITION` and `PR_TOOLBAR_Y_POSITION`
: Specify the screen coordinates of the upper-left corner of the floating toolbar. Double clicking on a floating toolbar at runtime docks the toolbar to the frame.

Changing the State of Toggle Items

Toggle items—on a menu and a toolbar—are initially set to the state specified in the menu script. Toggle items alternatively show or hide a system-specific indicator to show whether the item's state is on or off. If the toggle item is included in the toolbar, Panther uses its `MNI_ARM_PIXMAP` or `MNI_ACT_PIXMAP` property to show whether its state is on or off.

The function that you associate with a toggle item through its control string property should perform these tasks:

- Test the current setting of the item's `MNI_INDICATOR` property—set to either `PROP_ON` or `PROP_OFF`
- Execute the appropriate action.
- Change the item's `MNI_INDICATOR` property to `PROP_ON` or `PROP_OFF`.

For example, the following code examines the state of menu item `tgl2` and changes its `MNI_INDICATOR` property accordingly.

```
vars ind, ret
ind = sm_n_mnitem_get_int \
        (MNL_SCREEN, "toggle", "sub1", "tgl2", MNI_INDICATOR)
if (ind_state == PROP_ON)
{
  ret = tgl2_proc(PROP_ON)
  if ret > 0
  {
    call sm_n_mnitem_change_i_screen \
          ("toggle", "sub1", "tgl2", MNI_INDICATOR, PROP_OFF)
  }
}
else if (ind_state == PROP_OFF)
{
  ret = tgl2_proc(PROP_OFF)
  if ret > 0
  {
    call = sm_n_mnitem_change_i_screen \
           ("toggle", "sub1", "tgl2", MNI_INDICATOR, PROP_ON)
  }
}
```

Creating and Deleting Menus

- `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 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.

Inserting and Deleting Menu Items

- `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.
- `sm_mnitem_delete` removes an item from a menu and frees the memory associated with it. Panther updates the menu display at the next delayed write.

Uninstalling and Unloading Menus

Menus and their scripts remain in memory until Panther frees their memory location—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.

You can explicitly remove a menu from display by calling `sm_menu_remove`. This function takes a single argument that specifies the scope from which to remove the current menu. Because the menu script remains in memory, 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.

You can remove a script from memory with `sm_mnscript_unload`. This function takes two arguments—the script's name and memory location. Panther removes the 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.

Invoking Popup Menus

Panther displays a popup menu when the user presses the right mouse button or when `sm_popup_at_cur` is called. Panther uses one of the following two algorithms for finding and displaying a popup menu:

- If a field has focus, Panther displays the first menu that it finds from the following:
  - a. The field's popup menu.
  - b. The screen's popup menu.
  - c. The menu installed for the screen's menu bar and toolbar.
  - d. The application-level menu.
- If the screen has focus, Panther displays the first menu that it finds from the following:
  - e. The menu installed for the screen's menu bar and toolbar.
  - f. The application-level menu.

You can let users invoke popup menus from the keyboard with `sm_popup_at_cur`. For example, the following control string assignment lets the user invoke a popup menu by pressing the PF1 key:

```
PF1 = ^sm_popup_at_cur
```

Calling Menu Functions From JPL

All menu functions that can be prototyped are installed and can be called from a JPL procedure. However, three functions cannot be prototyped because their parameter lists do not conform to current requirements. These are:

- `sm_menu_change`
- `sm_mnitem_create`
- `sm_mnitem_change`

Wrapper functions for these routines are provided and installed in `funclist.c`; you can call these from JPL to change menu and menu item properties and to create menu items.

Outputting Menu Definitions to ASCII

You can save menu definitions to ASCII format through the `m2asc` utility. ASCII menu definitions define a menu as a hierarchy, where the top-level menu and its items are defined first along with global menu properties, followed by submenus and their items. You can edit the ASCII file using a text editor, and then convert it to binary format using the same utility. Refer to `m2asc` for an example of a menu file's ASCII output.

Keywords

Each component of a menu definition is identified by a keyword (refer to Table A-1 on page A-23 and, optionally, a unique name. In some cases, Panther uses these names to resolve references—for example, given a submenu item that sets its `SUBMENU` property to myEditSub, at runtime, Panther looks for a MENU:myEditSub item in the same script to build that submenu. In all cases, you can use these identifiers to get and set item properties at runtime.

Menu Properties

Each menu and menu item definition has properties; these properties are specified immediately below the component's identifier in the ASCII output. For example, the following statements define a submenu item myoption: its label is Options with a keyboard mnemonic of O; it invokes the menu myoptionsub; and it is initially available for selection (ACTIVE=YES):

```
SUBMENU:myoption
  LABEL=&Options
  SUBMENU=myoptionsub
  ACTIVE=YES
```

Table 15-1 lists all menu property mnemonics and their valid values. For additional information about the properties, refer to Chapter 25, “Menu Bar Editor,” in *Using the Editors*.

Table 15-1 Menu properties and valid assignments

Property	Values
ACCEL	Accelerator string that specifies the keyboard equivalent for selecting this menu item.
ACCEL-ACTIVE	Specifies whether the menu item accelerator is active (PROP_ON) or inactive (PROP_OFF).
ACTIVE	Allows (YES) or disallows (NO) user access to this menu item. If ACTIVE=NO, the menu item is greyed out.
ACTIVE_PIXMAP*	Name of image file whose contents are shown for active toolbar item—that is, accessible but not pressed. Refer to “Image File Types” on page 25-15 in *Using the Editor*s for valid file types, and for information about path and extension options.

Table 15-1 Menu properties and valid assignments *(Continued)*

Property	Values
`ARM-PIXMAP*`	Name of image file whose contents are shown for armed toolbar item—that is, in its pressed state. If the 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.
`CONTROL`	Control string that specifies the action that occurs when the item is selected.
`DISPLAY-ON`	Specifies whether to display the menu item on the menu and/or the tool bar. Supply one of these arguments: ■ `MENU`: Menu only (default). ■ `TOOL`: Tool bar only. ■ `BOTH`: Menu and tool bar. ■ `NEITHER`: Neither.
`EXTERNAL`	Specifies whether to find this menu's definition in another menu script (`YES`), or not (`NO`). External references are resolved at runtime only.
`EXT-HELP-TAG`	String expression that specifies the help text to invoke for this item.
`HOT-PIXMAP*`	In Windows, name of image file whose contents are shown when a mouse passes over the item.
`INACTIVE-PIXMAP*`	Name of image file whose contents are shown for an in active or unavailable (grayed) item. If blank, Motif displays an empty toolbar item; Windows uses a grayed version of the Active Pixmap property to display the item's inactive state and ignores this property.
`INDICATOR`	Specifies whether to show (`YES`) or hide (`NO`) the toggle indicator.
`IS-HELP`	Specifies whether to display (`YES`) this item as the right most item on the menu bar, or not (`NO`).
`LABEL`	String expression to display as menu item's label. To specify a keyboard mnemonic for a menu item, place an ampersand (&) in front of the desired character.
`MEMO`	String expression for the Memo Text property.

Table 15-1 Menu properties and valid assignments *(Continued)*

Property	Values
`MNI_ORDER*`	Order in which the item appears on the toolbar. Enter any value between 0 and 200 (default is 100), inclusive. If all toolbar items have the same value, they appear in the same order as they do in the menu.

`SEP-STYLE`	Style used by item separators, one of the following:
	`SINGLE`
	`DOUBLE`
	`DOUBLE-DASHED`
	`SINGLE-DASHED`
	`ETCHED-IN`
	`ETCHED-OUT`
	`ETCHED-IN-DASHED`
	`ETCHED-OUT-DASHED`
	`NOLINE`
	`MENUBREAK`

`SHOW-ACCEL`	Specifies whether the menu item displays (`YES`) or does not display (`NO`) the accelerator key next to the label.
`STAT-TEXT`	String expression to display on status line for this item.
`SUBMENU`	Name of submenu to invoke when the item is selected.
`TEAR`	Enables (`YES`) or disables (`NO`) the submenu as a tear-off menu.
`TITLE`	Title to display with tear-off submenus.
`TM-CLASS`	Transaction manager property. Refer to “Setting Classes for Menu Items and Push Buttons” on page 23-7 in *Using the Editors* for valid arguments.
`TOOL-TIP*`	Balloon help to display when the cursor remains over the toolbar item.

* *Ignored in character-mode.*

A subset of these properties is valid for each menu component except `WINOP` and `WINLIST`. Figure 15-1 shows which properties are valid for each component:

Property	Menu definition component					
	MENU	SUBMENU	ACTION	TOGGLE	SEPARATOR	ED*
ACCEL	•		•	•		•
ACCEL-ACTIVE	•		•	•		•
ACTIVE	•	•	•	•		
ACTIVE-PIXMAP			•	•		•
ARM-PIXMAP			•	•		•
CONTROL			•	•		
DISPLAY-ON			•	•		•
EXTERNAL	•					
HELP-TAG	•	•	•	•		•
INDICATOR	•			•		
INACTIVE-PIXMAP			•	•		•
IS-HELP		•	•	•		
LABEL		•	•	•		
MEMO		•	•	•		
ORDER			•	•		•
SEP-STYLE	•				•	
SHOW-ACCEL	•		•	•		•
STAT-TEXT		•	•	•		•
SUBMENU		•				
TEAR	•					
TITLE	•					
TM-CLASS		•	•	•		
TOOL-TIP			•	•		•

* All editor item types: EDCUT, EDCOPY, EDPASTE, EDDEL, EDSELECT, EDCLEAR

Figure 15-1 Menu components and their properties

CHAPTER

16 Building Reports

With Panther, you can build reports to supplement your application processing. Once created, a report can be invoked from an application, from a web browser, or on the command line, and can be output to the screen, to a printer or to a file.

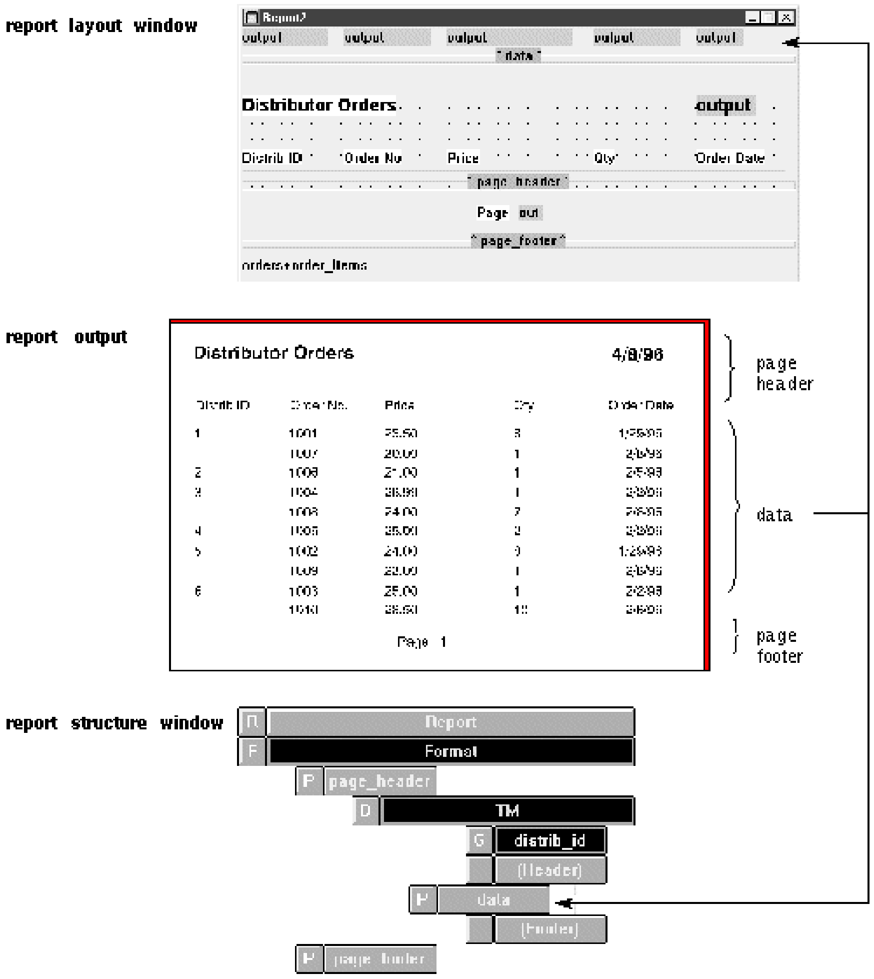
A report definition has two windows in the editor: the report layout window and the report structure window.

The report layout window, containing one or more layout areas, defines the report content. Each layout area contains widgets whose properties define the source of report data. A widget's position in the layout area determines its position in the report output.

The report structure window, consisting of a series of nodes, determines the order of report processing. Each layout area must have a corresponding print node in the report structure in order to appear in report output. Other nodes define the format of the report, the properties of report groups, and the programming actions to take during report processing.

Figure 16-1 illustrates the two report windows, report layout and structure, and how those parts of the report definition combine to produce the report output.

For information on creating and customizing reports, start with Chapter 1, “Overview of Panther Reports,” in *Reports*.



```
Distributor Orders                                    4/8/96

Distrib ID    Order No.    Price          Qty        Order Date

1             1001         25.50          3          1/25/95
              1007         20.00          1           2/6/93
2             1006         21.00          1           2/5/93
3             1004         26.90          1           2/8/93
              1008         24.00          7           2/6/95
4             1005         25.00          2           2/8/93
5             1002         24.00          3          1/29/93
              1009         22.00          1           2/6/93
6             1003         25.00          1           2/2/93
              1010         26.50          10          2/8/93

                          Page  1
```

Figure 16-1 The report layout window and the report structure window work together to define a report.

Part IV Preparing the Programming Interface

After creating your application components, you need to program your application behavior. This section explains Panther's application events and how to call C, Java and JPL functions in your application. Later chapters describe how to specify the screen sequence and move data between screens.

CHAPTER

17 Understanding Application Events

Almost everything that happens during the life span of a Panther application is identified by Panther as one type of event or another. At one level, events occur as the result of user interaction with the interface—for example, pressing a push button widget, or double-clicking on a dynamic label. These events and the control strings that determine the application's response are discussed in Chapter 18, “Programming Control Strings.”

This chapter initially focuses on application events that are not always connected to user actions. For example, a Panther application always starts by opening a screen. Whenever a screen opens, the same sequence of events occurs, such as initialization of the transaction manager and execution of the screen's entry function. Functionality can be associated with an event, such as screen opening, by means of various properties or by installing C or Java functions.

Your ability to control application behavior largely depends on knowing the order in which events occur and what options, or hooks, are available that allow you to intervene. This chapter describes events that typically occur during a Panther application, and the hooks that are available for each one. For purposes of this discussion, events are broadly grouped according to the following application components:

- Screens and widgets
- Transaction manager
- Database interface
- Web

- Middleware API

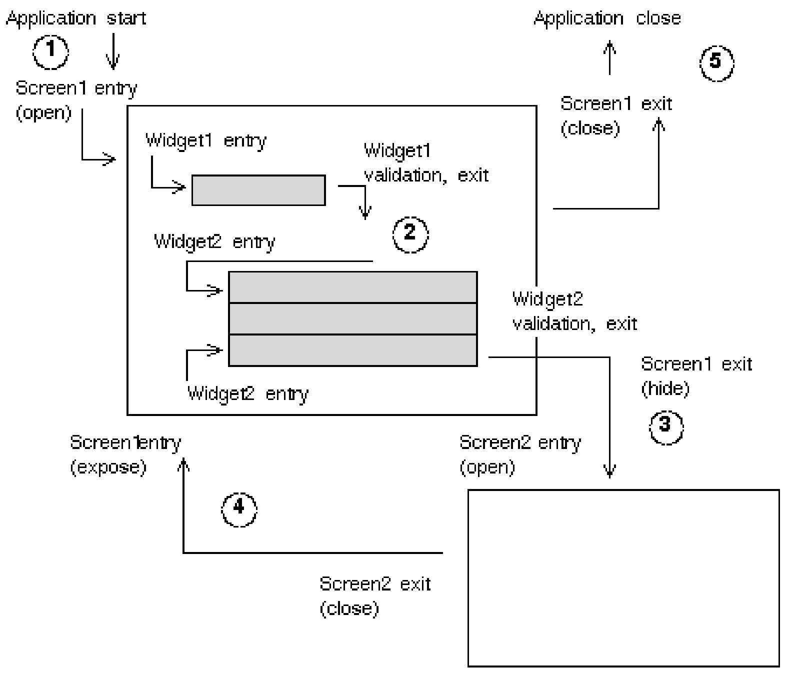
The final section of the chapter discusses these event categories in relation to user-initiated actions.

This chapter assumes that you are already familiar with the basics of Panther application development, as covered by Chapter 1, "Building a Panther Application" and the tutorial in *Getting Started 2-Tier* or *Getting Started JetNet/Oracle Tuxedo.*

Screen and Widget Events

Screens and widgets provide the foundation of a Panther application. All user interaction takes place via a client screen and its widgets; and application code is typically accessible, either directly or indirectly, through client screens or, in three-tier applications, the service components that they invoke. Thus, much of the processing that takes place in a Panther application occurs at the screen and widget level, and can be decomposed into distinct screen and widget events.

The following drawing schematically depicts screen and widget events as they occur in an application that consists of two screens, Screen1 and Screen2:



Application workflow and accompanying events consist of these steps:

1. Screen1 opens on application startup; this triggers screen entry and widget entry events, as focus is given to the first widget on the screen, Widget1.
2. When the user tabs out of Widget1, Panther performs validation and exit processing for Widget1, then executes widget entry for Widget2.
3. The user opens Screen2 while Widget2 has focus. Screen2 is opened as a stacked window, so Screen2 remains visible but is focus-protected. This action causes several events:
   - Validation and exit processing for the third occurrence in Widget2—Widget2[3]
   - Screen exit for Screen1
   - Screen entry for Screen2
   - Widget entry for the first widget on Screen2

4. When the user exits from Screen2 and returns to Screen1, these events occur:
   - Validation and exit processing for the Screen2 widget that has focus
   - Screen exit for Screen2
   - Screen entry for Screen1
   - Widget entry for the last occurrence on Screen1 to have focus, Widget2[3]
5. When the user exits the application from Screen1, Panther performs validation and exit processing for the widget that has focus, then screen exit for Screen1.

Most screen and widget events can be monitored and controlled programmatically and through Panther properties. Functions that execute on events are called event functions. An event function can be installed in the application as an automatic function, which executes on all events of a given object type (screen or widget); or it can be named in an event-specific property of a given screen or widget, so it executes only for that object on a specific event. For example, an automatic screen function executes on all entry and exit events for every screen in the application; however, a function that is named in a screen's Entry Function property executes only when that screen opens or regains focus.

Properties can also be used to control validation processing or check a widget's status. For example, you can set a widget's required property to force user input; and you can check whether a widget's data has changed through its runtime mdt property.

Getting your application to behave as you wish depends on knowing the sequence in which screen and widget events occur and what options are valid for each one. The following sections discuss these events in the order of occurrence. In addition, the Panther debugger can be used to determine the sequence of events for a particular screen.

Screen Entry

Screen entry occurs when a screen opens or regains focus. Because an application begins by opening a screen as its base form, the processing that is associated with screen open provides the first—and, often, the most important—set of opportunities to determine an application's behavior. Screen exposure occurs when a screen regains focus from a screen that previously overlay it. The set of events that occur on screen exposure is a subset of the set of events that occur on screen open.

Open Events

When a screen opens, events occur in the following order:

1. The screen's unnamed JPL procedure executes. (A `public` or `include` statement in this procedure also executes that module's unnamed JPL procedure.)
2. Transaction manager is initialized.
3. Automatic screen function executes.
4. Screen entry function executes.
5. LDB write-through occurs (if there's an open LDB).
6. `valided` and `mdt` properties are cleared (set to `PV_NO`) for all data input widgets.
7. Automatic group, grid or tab control function executes, if the first field on the screen is in a container.
8. Group, grid or tab control entry function executes, if the first field on the screen is in a container.
9. Automatic field function executes for the first field on the screen.
10. Field entry function executes for the first field on the screen.
11. Pushed keys in input stack are processed.
12. Keyboard opens, screen displays.

All processing that should occur before the user sees the screen must take place during these steps. The nature of the processing itself dictates the specific step at which it should occur. For example, transaction manager commands can be issued only after the transaction manager is initialized (step 2).

Always allow all steps on this list to complete. Specifically, avoid opening a screen while opening another screen; doing so can prevent the original screen from completing its open processing and yield unpredictable results.

Exposure Events

At different times during an application, the active window can open another screen as a sibling or stacked window; the newly opened window overlays its caller. When the original window regains focus, screen exposure processing takes place. Screen exposure processing also occurs when a report returns to the invoking screen (refer to the `runreport` command).

Screen exposure processing is a subset of open processing, and consists of these steps:

1. Automatic screen function executes.
2. Screen entry function executes.
3. LDB write-through occurs.
4. Automatic field function executes (for the field that gains focus).
5. Field entry function executes (for the field that gains focus).
6. Pushed keys in input stack processed.
7. Keyboard opens, screen is redrawn.

When a screen is exposed, focus returns to the last field on the screen to have had focus. If this field is in a group, grid or tab control, there will be entry events associated with the container prior to the automatic field function.

A screen's unnamed procedure executes and the transaction manager is initialized only once for a given screen, when it opens. Also, the `valided` and `mdt` properties for the screen's data input widgets are cleared (set to `PV_NO`) only when a screen opens. Subsequent exposures of the screen leave these properties unaffected.

Unnamed Procedure

A screen's JPL (accessed through its JPL Procedures property) is accessible to the entire screen. When the screen initially opens, its unnamed procedure—the code that precedes the first `proc` statement—executes first.

A screen executes its unnamed procedure only when it opens; subsequent exposures of the screen ignore this code. Thus, the unnamed procedure of an application's opening screen is the appropriate place to make public those JPL library modules that are required by the entire application or make other JPL modules available for this screen.

The `public` command makes these modules' procedures accessible to this and other application screens and their widgets. A public module's procedure can be named in screen or widget properties—for example, in a widget's Entry Function property—or called from other JPL procedures with the `call` command.

When Panther loads a public module, it loads the module's procedures into memory and executes its unnamed procedure, if any. During its life span, an application can call any named procedure in a public module, unless the module is explicitly removed from memory with the `unload` command.

A public module's unnamed procedure executes only once; and a module can be made public only once. (Panther ignores any public command that is issued on a module that is already public.) Thus, any code in a public module's unnamed procedure is virtually guaranteed to execute only once during the life span of an application.

The `include` command makes a modules' procedures accessible to the current application screen.

In general, it is good programming practice to public only the procedures that you need to be global to the entire application. Procedures specific to individual screens should be in modules that are included by the screens to which they refer.

It is suggested that you maintain JPL code in libraries for the following reasons:

- Library JPL is independent of a given screen binary, so it is available to any application and its screens.
- Library JPL can be put under source control. (Refer to page 10-4, “Maintaining Libraries Under Source Control.”)
- It is easier to locate and update library JPL than code that is embedded in a screen or widget.
- For JetNet/Oracle Tuxedo applications, if the library resides on a file access server, its JPL is accessible to other Panther developers, and subject to the safeguards of Panther libraries. (Refer to Chapter 10, “Accessing Libraries.”)

Transaction Manager Initialization

The transaction manager is initialized only once for a given screen. You can issue transaction manager commands such as `SELECT` (via `sm_tm_command`) only after this step is complete. Thus, it is an error to issue this or any other transaction manager command in a screen's unnamed procedure. Use a later stage of screen open processing to issue these commands—for example, the screen's entry function.

To disable transaction manager initialization, set the screen's Root property to `None`.

Screen Automatic and Entry Functions

A screen's automatic function and entry function are called successively on screen entry:

- The automatic screen function written in C must be specifically installed via the source file `funclist.c`. This function is called automatically on all screen entry and exit events. For more information about installing automatic screen functions, refer to page 44-13, "Installation of an Automatic Screen Function."

  If the screen function is written in Java, the Java Tag property must specify either a screen-specific Java class, which would access a corresponding Java function, or the default screen Java class, which would access the screen entry processing in the current class factory. For more information about writing Java functions, refer to Chapter 21, "Java Event Handlers and Objects."

- The screen entry function is explicitly named in the screen's Entry Function property. You can write screen entry functions in JPL, in C, or in Java. If written in C, a screen entry function must be installed either as a demand screen function (refer to page 44-13, "Installation of Demand Screen Functions") or as a prototyped function (refer to page 44-9, "Installing Prototyped Functions"). Because they are explicitly named, you can write multiple entry functions that are individually tailored to specific screens. Screen entry functions are typically used for processing that should occur every time a screen opens or is exposed.

  To write screen functions in Java, you must specify an event handler class for the screen in the Java Tag property. If a screen has both the Java Tag and Entry Function properties specified, both types of processing will occur; first the Java event handler, then the named function.

Panther always executes the automatic and named entry functions on screen open. It also executes these functions on screen exposure if the setup variable `EXPHIDE_OPTION` is set to `ON_EXPHIDE` (the default). If so, the screen's automatic and entry functions are the first events to occur on screen exposure.

Panther automatically supplies two arguments to screen functions: the screen's name; and an integer bitmask that indicates the screen's current state and why the function was called. (For information on using these arguments in a JPL procedure, refer to page 19-3, "Passing Standard Arguments to JPL Procedures.") Screen functions can use these arguments to control execution that is specific to a given screen; or (more typically) to distinguish between screen event types.Panther provides three mnemonics that can be bit-wise AND'ed with the context argument to determine which screen event caused the screen function to be called:

- `context & K_ENTRY` — screen entry
- `context & K_EXIT` — screen exit
- `context & K_EXPOSE` — screen hidden or exposed
- `context & (K_EXPOSE | K_APP_FOCUS)` — Windows application has lost or gained focus. This only happens when the property `@app()->hook_app_focus_change` is set to `PV_YES` (Panther 5.20 and later).

For example, the following screen function code tests whether the function is called on screen open or screen exposure:

```
int scr_entry_func( char *scr_name, int context )
{
    if( context & K_ENTRY )
    {
        if( context & K_EXPOSE )
        {
            // expose processing
        }
        else
        {
            // open processing
}   }   }
```

The following example tests if the screen is being opened for the first time, instead of just being exposed:

```
int scr_entry_func( char *scr_name, int context )
{
    if !( context & K_EXPOSE )
    {
            // only the first screen open processing
}   }
```

LDB Write-through

Panther screens can be used as vehicles for initializing and saving values on other screens. A screen that performs this background role is called a local data block, or LDB. When a screen serves as an LDB, Panther uses its data input widgets, or LDB entries, to transfer data to and from corresponding widgets on the current screen. Panther matches LDB entries and screen widgets by name. By using LDBs, applications can transfer data between screens automatically.

LDB write-through occurs on both screen open and expose events. No hook is provided to intervene in this process; however, it is important to know that any attempt before this step to set a screen widget's value is actually written to the corresponding LDB entry and so might overwrite any previous value in it. Note also that on screen open, the LDB respects initial widget data that is specified through the screen editor—for example, the value set in a push button's Label property. Initial widget data also is written to the corresponding LDB entry.

For more information about LDB processing, refer to page 25-7, "Using Local Data Blocks."

Clearing of Valided and Mdt Properties

Panther provides two widget-level runtime properties, `valided` and `mdt`, which are automatically cleared (set to `PV_NO`) on screen open for all data input widgets. In other words, when the screen first opens, all data input widgets are regarded as requiring validation and as unmodified.

A widget's `valided` property remains clear until it passes validation. This sets the `valided` property to `PV_YES` until the widget's data changes again.

A widget's `mdt` property remains clear until its data changes. Thereafter, the widget's `mdt` property is set to `PV_YES` unless it is explicitly cleared, either individually or with all data input widgets on the screen by a call to `sm_cl_all_mdts`.

No hook is provided to intervene in this processing; however, it is important to know that Panther sets these properties after the screen's automatic and entry functions execute and LDB write-through occurs—in other words, after the stages in which widget data is typically initialized, either from a database or LDBs. So, Panther marks as potentially invalid widgets whose initial data is probably valid. By testing both the `valided` and `mdt` properties, you can determine whether a widget actually requires validation.

For more on using the `valided` and `mdt` properties, refer to "Field Validation" on page 17-14 and "How to Validate All Screen Widgets" on page 17-20.

Field Automatic and Entry Functions

The automatic field function and a field's own entry function are called successively on field entry. The automatic field function is called on all field events; a field's entry function is named in the field's Entry Function property and is there fore specific to that field and event. Field entry functions are typically used for processing that should occur every time a given field gains focus.

For more information about automatic field functions, refer to “Field Functions” on page 44-14; for more about field event functions, refer to “Event-specific Functions and Arguments” on page 17-13.

Input Stack Processing

Just after Panther opens the keyboard and before it displays the screen to the user, it checks the input stack for unprocessed keys; if any are found, it pops them off the stack and processes them. You can push keys onto the input stack earlier during screen entry by calling `jm_keys`. Because application keys can be associated with control strings through a screen's Control Strings property, the input stack can be used to defer any processing that cannot safely be called earlier during screen entry, such as calls to open a screen.

For example, on application startup you might want to display a login screen that appears simultaneously with the application's base screen and overlays it. As noted earlier, a screen should never try to open another screen until its own open processing is complete. You can implement this behavior in two steps:

1. In the base screen's Control Strings property, attach the appropriate control string to an unused application key. For example:

   ```
   APP1 = &login.scr
   ```

2. At an appropriate stage of the base screen's entry processing, enter this statement:

   ```
   call jm_keys APP1
   ```

   Because a login screen appears only at application startup, this call should be made at a stage of the base screen's entry processing that is not repeated on later exposures of the same screen. The unnamed procedure of a module that the base screen makes public (in its own unnamed procedure) executes only once, so it should issue the call to open the login screen.

Now, when the application starts and opens its base screen, the call to `jm_keys` pushes APP1 onto the keystack, where it remains until the keyboard opens; Panther pops APP1 off the keystack and in doing so executes APP1's control string and opens the login screen as a stacked window. This done, Panther draws the display and shows both screens as if they opened simultaneously, with login.scr as the topmost, active window.

Frameset Events

Frameset event processing is described in “Cursor Movement and Window Management using Framesets” on page 17-7 in *Using the Editors*.

Widget Events

Fields, or data entry widgets, can undergo three events: entry, validation, and exit. Dynamic labels are the exception to field event processing; since dynamic labels cannot be entered, they only have validation events.

The following widget types are not fields, so other exceptions in event processing apply:

- Widget types that have no data content and cannot gain focus have no event processing: boxes, lines, static labels, graphs, links, and table views.
- A grid widget or a selection group have no data—the data that appears in these containers actually belongs to their member widgets—so it is not subject to validation; however, these containers can be entered and therefore have entry and exit events. In addition, grid widgets have row entry and exit events.
- Tab cards are not fields; however, the tabs that can be used to select a given card are fields and support entry, validation and exit events.

Panther provides various properties that let you control entry, validation and exit processing; properties that are specific to each event are described in later sections. All three events share these features:

Automatic Field Function

All fields initially call the automatic field function if one is installed. The automatic field function must be written in C, and it must be installed via the source file funclist.c. For more information about installing an automatic field function, refer to page 44-14, “Field Functions.”

Event-specific Functions and Arguments

Almost all widgets provide properties that let you name functions to execute on specific events—Entry Function, Validation Func, and Exit Function. Some widget types have a Control String property, which can name a function to execute on validation events.

The functions that you name in these properties can be written in JPL, in C, or in Java. If written in C, the function must be installed either as a demand field function (applicable only to data input widgets or as a prototyped function. Because they are specified using individual properties of a given widget, you can write functions that are tailored to specific widgets or events. For example, a function that should execute every time a widget gains focus should be attached to that widget's Entry Function property. For more information, refer to “Installation of Demand Widget Functions” on page 44-20 and “Installing Prototyped Functions” on page 44-9.

If written in Java, the Java Tag property must specify the Java event handler. If a widget has both the Java Tag and Entry Function properties specified, both types of processing will occur; first the Java event handler, then the named function. For more information, refer to Chapter 21, “Java Event Handlers and Objects.”

Panther automatically supplies four arguments to any event function that is specified for a data input widget:

- Field number.
- Field value.
- Occurrence number.
- An integer bitmask that indicates the field's validation state and why the function was called.

Panther supplies these arguments if the event function property contains only the function's name. The function can use these arguments to control execution that is specific to a given widget, or to identify widget event types—for example, to differentiate between close and hide events.

Panther also supplies arguments for grid widgets, selection groups, and tab cards. For more information, refer to page 19-21, "Calls from Screens and Widgets."

Field Entry

When a screen opens or is exposed, the first field that can gain focus undergoes entry processing. Thereafter, a field gains focus either because of user action such as tabbing or mouse-clicking into it, or through programmatic manipulation—for example, a call to `sm_tab` or `sm_gofield`.

Entry processing for a field can comprise two stages: first, the automatic field function (if one is installed) executes; then the field's own entry function executes. A field's entry function is a named function that you set in the field's Entry Function property. This property is available for all data input widgets such as single line text; it is also available for grid widgets and selection groups.

Field Validation

When a field loses focus or it is selected or deselected, it undergoes validation processing unless its `no_validation` property is set to `PV_YES`; the default is `PV_NO`. Field validation can occur as a result of various events or actions, including the following:

- The user moves the cursor out of the field by pressing the Tab or Enter key, or fills a field that has its Autotab property set to `PV_YES`.
- The user selects or deselects a selection widget, such as a radio button, or one of the items in an action list box. Selecting an item in an action list box where another item is already selected causes both the deselected item and the selected item to undergo validation.
- You programmatically force validation for a given field by calling `sm_fval`, or for all fields and their occurrences on a screen by calling `sm_s_val` or `sm_validate`. You typically call `sm_s_val` or `sm_validate` in a screen's exit function, or when you attempt to save screen data.

Avoid Unnecessary Validation

Field validation always occurs on the aforementioned events regardless of the widget's `valided` property setting. You can avoid unnecessary processing by checking the widget's validation state in any function that executes during validation. This JPL function tests the current field's `valided` property:

```
proc wdg_valid_func()

if ( @widget("@current")->valided )
{
   return
}
else
{
   //do validation processing
}
```

As mentioned earlier, Panther sets all widgets' `valided` property to `PV_NO` on screen open; inasmuch as a screen's entry function might initialize these widgets from database values, the `valided` setting might not reliably tell whether a widget's data is valid. In this case, you can also test a widget against its `mdt` property setting:

```
proc wdg_valid_func()
vars valid = @widget("@current")->valided
vars modified = @widget("@current")->mdt

if ( valid || !modified ) //data either valid or unchanged
{
   return //no further processing required
}
else
{
   //do validation processing
}
```

Note: You can also test a widget's validation and modified states by bit-wise AND'ing the VALIDED and MDT mnemonics with the fourth standard argument received by widget functions, as in this code:

```
proc wdg_valid_func( fldno, data, occ, context )
if( context & VALIDED || !context & MDT )...
```

Properties Tested During Validation

During field validation, Panther tests a field's data against a number of formatting and input property settings, in the order shown in the following table. Some fields 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 17-1 Properties tested during field validation

Property setting	Skip if valid	Skip if empty
`required = PV_YES`	y	n
`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; refer to "Defining a Numeric Format" on page 10-20 in Using the Editors.*

The JPL in a field's Validation Func and JPL Validation property must return 0 to indicate success. Other values indicate failure: a return value of 1 leaves the cursor at its last position; any other non-zero value repositions the cursor at the widget's first position.

Non-validation Events

Field validation does not typically occur when the user uses a cursor key to move out of the widget, or mouse clicks into another widget. To force validation also to occur on these events, set the application setup variable IN_VALID to OK_NOVALID. Because users expect to move freely within a GUI application screen, validation is typically suppressed until they explicitly submit the screen data—for example, by pressing a Save push button. Therefore, IN_VALID is by default set to OK_NOVALID.

Field Exit

Exit processing occurs for a field that has focus when the user moves the cursor out of the field, or the screen closes or is hidden. All methods of moving the cursor out of the field, including mouse clicking, trigger exit processing. If the exit event is also one that requires validation processing, exit processing begins only when the widget's valided property is set to PV_YES.

Exit processing for a field can comprise two stages: first, the field's own exit function executes; then the automatic field function (if one is installed) executes. A field's exit function is a named function that you set in the field's Exit Function property. This property is available for all data input fields such as single line text; it is also available for grid widgets and selection groups.

Grid Column Label Click

When a widget is a grid member, what happens when its column label is clicked depends on its Click Column Action property, which can be None, Sort or Custom.

When Click Column Action is Sort, the Sort Order should be set. The order can be Lexicographic for an alphabetic sort; Numeric for a numeric value sort; Date/Time for sort by date and time; and Custom to allow you to use your own sort function. The Sort Order Func property names this function. It can be written in JPL or can be installed as a prototyped function. The function is passed two strings to compare. The return value should be COMPARE_LESS if the first string is less than the second, COMPARE_EQUAL if the two strings are equal, COMPARE_GREATER if the first string is greater than the second or COMPARE_ERROR if an error has occurred.

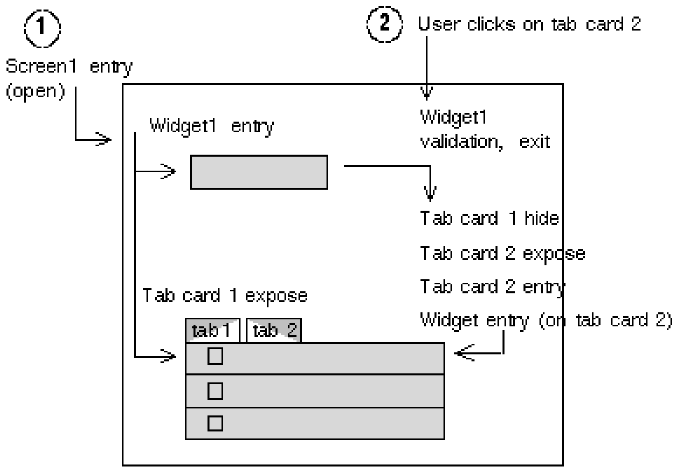
When Click Column Action is Custom, a Column Click Func can be specified that will be called when the widget’s column label is been clicked. This function can be written in JPL or can be installed as a prototyped function. The function is passed the object ID of the widget. Its return is ignored.

Tab Control Events

Events occur for the index tab on a tab card and for the tab card itself; the tab deck has no events. Cards in a tab deck have expose and hide events along with entry and exit events.

A tab card expose event occurs when a card becomes the topmost card in the deck and during screen entry for the topmost card in the deck, even if the tab deck is hidden. A hide event occurs when another card on the deck becomes the topmost card, when the card is deleted, or when the screen is closed.

When moving to a new topmost card, a card exit event will not precede a card hide event if a widget outside the tab deck has focus. The following drawing schematically depicts the events that would occur in this context:



Panther provides three mnemonics that can be bit-wise AND'ed with the context argument to determine which event caused the tab card's function to be called:

- context & K_ENTRY — tab card entry
- context & K_EXIT — tab card exit
- context & K_EXPOSE — tab card exposed

If all three bits are clear, the tab card hide event was called. For more information, refer to page 44-24, "Tab Control Functions."

Screen Exit

Screen exit occurs when a screen closes or is hidden by another screen. Because an application typically ends by closing the screen that is its base form, exit processing for this screen can perform any cleanup that the application requires, such as closing database or middleware connections. Screen hide events occur when a screen loses focus to another screen, or invokes a report by calling the runreport command. Screen hide events are a subset of close events.

When a screen closes or is hidden, the following events occur:

1. Exit processing for the current widget, as described on page 17-17 under "Field Exit."

2. LDB write-through. All widget data is written to corresponding entries in active LDBs.

3. Screen exit function executes. The screen exit function, like the screen entry function, is a named function; specify the function in the screen's Exit Function property. Screen exit functions are typically used for processing that should occur every time a screen is hidden or closed.

   The functions that you name in these properties can be written in JPL, in C, or in Java. If written in C, a screen entry function must be installed either as a demand screen function or as a prototyped function. Because they are explicitly named, you can write exit functions that are individually tailored to specific screens.

   To write screen functions in Java, you must specify an event handler class for the screen in the Java Tag property. If a screen has both the Java Tag and Entry Function properties specified, both types of processing will occur; first the Java event handler, then the named function. Panther automatically supplies the same two arguments to screen exit functions that it supplies to screen entry functions.

4. Automatic screen function executes as described earlier. This function must be written in C.

5. Transaction manager closes (only on screen close).

Note: Panther always executes the screen's automatic screen function and the screen's named exit function on screen close. It also executes these functions on screen expose if setup variable `EXPHIDE_OPTION` is set to `ON_EXPHIDE` (the default).

Screen Exit Processing

Screen exit processing often tests widget data, either by forcing validation processing or by testing whether widget data has changed. However, you should not open another screen during screen exit.

How to Validate All Screen Widgets

Call `sm_s_val` or `sm_validate` which traverses the screen, testing each widget occurrence and setting its valided property to `PV_YES`. If it encounters an occurrence with invalid data, the function returns, posts an error message, and positions the cursor there. The valided property remains unchanged for all untested widgets.

How to Test Screen Data Changes

Call `sm_tst_all_mdts` which checks `the` mdt property for each data input widget and its occurrences. When the function encounters an occurrence that contains modified data, it returns with information that identifies the widget and occurrence number. For example, the following function alerts users to changed data on the current screen:

```
int dataChanged()
{
    int fldno, occ, wdg;
    char *nm;

    fldno = sm_tst_all_mdts( &occ );
    if( fldno >= 0 )
    {
        // get the widget's name
        wdg = sm_prop_id( "@field_num(fldno)" );
        nm = sm_prop_get_str( wdg, PR_NAME );
        if ( sm_message_box
              ( "New data in "##nm##" field--Continue?",
                "", SM_MB_YESNO ) == SM_IDNO )
            return sm_o_gofield( fldno, occ );
    }
    return 1;
}
```

Because `sm_tst_all_mdts` is not callable from JPL, you must either use it in a C function, or write a wrapper function that you install as a prototyped function and call from JPL. Alternatively, you can write JPL code that traverses all screen widgets and test each one's mdt property, as in this example:

```
proc tst_mdts()

vars nextObj, answer, wdg, ctObjs, occ, wdgOccs, nm
vars objList = sm_list_objects_start( @screen("@current")->id )

if( objList > 0 )
{
   // get the number of widgets on screen
   ctObjs = sm_list_objects_count( objList )

   // start traversal
   for wdg = 1 while wdg <= ctObjs
   {
      nextObj = sm_list_objects_next( objList )
      nm = @id( nextObj )-> name

      // get all the occurrences of this widget
      wdgOccs = @id( nextObj )-> num_occurrences

      // test mdt property of each widget occurrence
      for occ = 1 while occ <= wdgOccs
      {
        if( @id( nextObj)[occ]-> mdt )
        {
           if ( sm_message_box \
              ("New data in "##nm##" field--Continue?",\
                  "", SM_MB_YESNO) == SM_IDNO )
           {
              call sm_o_gofield \
                (@id( nextObj )->fldnum, occ)
              return 0
}   }   }   }   }
return 1
```

Closing Screens

The following actions close a screen:

- Choosing Close on the system menu.
- Calling `jm_exit` in a function or control string.

  If you call `jm_exit` in a JPL procedure, no other JPL programming should follow that statement. Otherwise, this subsequent programming will execute on the underlying screen in the stack.
- Pressing the `EXIT` key with its default setting.

  By default, the `EXIT` logical key is set to call `jm_exit`, which means that pressing Esc on most terminals closes the screen. However, if a control string is defined for the `EXIT` key, Panther executes the control string instead of calling `jm_exit`.

Exiting an Application

You can exit an application by:

- Closing the final screen in the form stack.
- Pressing the `CLAPP` key.

`CLAPP` is the Panther logical key that closes an application. For example, the following control string attached to the final screen calls `jm_keys` to issue the `CLAPP` key and a Y response to the termination message:

```
^jm_keys CLAPP 'Y'
```

Programming User-initiated Events

Table 17-2 lists common user-initiated events and the associated event hooks where application processing can be specified. This table references many widget events described on page 17-12, under “Widget Events.”

Table 17-2 Typical user events and their associated event hooks

Key/mouse events	Best event hook	Alternate event hooks
User enters grid	Grid Focus→Entry Function property	Widget Focus→Entry Function property of a grid member Widget Validation→Control String property of a listbox in the grid Widget Validation→Validation Function property of a listbox in the grid
User enters text widget	Widget Focus→Entry Function property	Default field function
User enters text	keyfilter routine	logical keys that are mapped to physical keys
User clicks on push button	Widget Validation→Control String property	Widget Validation→Validation Function property Widget Focus→Entry Function property
User enters list box (Select Any)	Widget Focus→Entry Function property	Group Validation→Validation Function property of listbox' group Group Focus→Entry Function property of listbox' group
User enters list box (Action)	Widget Validation→Control String property	Widget Validation→Validation Function property Widget Focus→Entry Function property
User double clicks	Widget Validation→Double Click property	Screen Focus→Control String property—MOUS/MDBL

Table 17-2 Typical user events and their associated event hooks *(Continued)*

Key/mouse events	Best event hook	Alternate event hooks
User leaves field	Widget Focus⟶Exit Function property	Widget Validation⟶Validation Function property Widget Focus⟶Entry Function property of next/previous field
User leaves grid	Grid Focus⟶Exit Function property	Widget Focus⟶Exit Function property of a grid member
User clicks grid column	Widget Format/Display->Sort Order Func	Widget Format/Display->Column Click Func
User does nothing	timeout function	timer function

Transaction Manager Events

The transaction manager generates a series of events in order to fetch database data, track any modifications to that data, and update the database with any modifications. The events are divided into two levels: requests and slices.

When you call a transaction manager command, such as `VIEW` or `SAVE`, the transaction manager generates that command's requests. For the `VIEW` command, the request events would be `PRE_VIEW`, `VIEW`, and `POST_VIEW`. The transaction manager then accesses the transaction model to determine what processing to perform for each request. Since a single request could have several steps, each request can be sub-divided into slices. The transaction model also determines the processing necessary for each slice.

For more information on transaction manager operations, refer to Chapter 31, “Building a Transaction Manager Screen.” For descriptions of transaction manager commands and their requests and slices, refer to Chapter 8, “Transaction Manager Commands,” in *Programming Guide*.

Database Interface Events

Database requests can generate errors that are reported by Panther's database drivers and the database engine itself. Panther provides global variables and hook functions that can help identify and manage these errors. Default error handlers are installed to report errors from Panther's database drivers and from the database engine. You can also write and install your own error handlers. With the JetNet/ Oracle Tuxedo middleware adapter, these errors can be logged to your application server.

For full information about database error handlers, refer to Chapter 37, “Processing Application Errors.”

Web Application Events

The online *Web Development Guide* describes the events that occur at runtime for a Panther web application. For more information, refer to Chapter 5, “Web Events,” in *Web Development Guide*.

You can also include JavaScript or VBScript programming in your client screens for events occurring in the web browser. For more information, refer to Chapter 9, “Using JavaScript and VBScript,” in *Web Development Guide*.

Middleware Events

In three-tier applications, the type of middleware determines the specification and type of events.

JetNet and Oracle Tuxedo Events

In JetNet and Oracle Tuxedo applications, events have been defined for service requests and as those events occur, they are forwarded to handlers for processing.

Panther provides built-in event handlers for all middleware event types. You can also write and install your own handlers in JPL or C. These handlers can perform all required processing on their own, or they can call the built-in handlers and overlay these with desired enhancements.

The following table lists event types for JetNet and Oracle Tuxedo applications:

Event type	Description
`advertise`	A service has been advertised
`exception`	An error or unusual change in the normal flow of program execution
`JIF_changed`	The JIF has been changed
`message`	A client receives an unsolicited message
`post_request`	A service request is completed
`post_service`	A service completes execution
`pre_request`	A service request is initiated
`pre_service`	A service is about to begin execution
`request_received`	A service request is received by the server

Event type	Description
`server_exit`	A server is brought down in an orderly fashion
`unadvertise`	A service has been unadvertised
`unload`	Data is received from an external source that can be written (unloaded) to Panther variables

Middleware events in JetNet and Oracle Tuxedo applications occur independently of screen and field events and transaction management events; however, their handlers can initiate actions that themselves precipitate events of these types. For example, an exception handler might respond to an error by aborting a service request, which in turn causes the transaction manager to report an error.

For more information about JetNet/Oracle Tuxedo event types and their handlers, refer to Chapter 6, "JetNet/Oracle Tuxedo Event Processing," in *JetNet Guide/Oracle Tuxedo Guide*.

CHAPTER

18 Programming Control Strings

User interaction with an application generally consists of entering data and making choices with function keys and menus, and with action widgets such as push buttons. You associate actions with these application objects through control strings.

Control strings can perform these tasks:

- Open a screen as a form or window.
- Execute a function.
- Invoke an operating system command.

For a table for user-initiated events, refer to Table 17-2 on page 17-23.

Associating Control Strings with the Application

A Panther application has various hooks from which it can execute control strings. You can associate control strings with push buttons, list boxes, and menu items through their Control String property. Some widget types can be double-clicked on; their Double Click property also takes a control string: dynamic labels, single- and multiline text widgets, list boxes, and combo boxes. For example, a push button widget specifies to exit the current screen when it is pressed if its Control Strings property is set to execute the built-in function `jm_exit`:

```
^jm_exit
```

You can also attach control strings to function keys. Each screen has its own Control Strings property, which lets you list Panther logical keys and a corresponding control string. For example, the following control string list lets users open two screens as windows through the logical keys PF1 and PF2, and leave the current screen through `EXIT`:

```
PF1     = &custInfo
PF2     = &orderDetail
EXIT   = ^jm_exit
```

You can globally associate control strings with function keys at application startup through the setup variable `SMINICTRL`. For example, the following statement in your setup file globally associates the `EXIT` key with your own exit routine:

```
SMINICTRL = EXIT = ^myExit
```

Control String Types

Panther uses the leading character of a control string to determine what type of action to perform—whether to open a screen, execute a function, or invoke a system command. Table 18-1 summarizes these leading characters and actions.

Table 18-1 Control string types and leading characters

Character	Action	Example
`None`	Open screen as a form.	`mainmenu`
`&`	Open screen as a stacked window.	`&(5,20)status`
`&&`	Open screen as a sibling window.	`&&(5,20)status`
`^`	Execute C function or JPL procedure.	`^drop acctno`
`!`	Invoke operating system command.	`!ls "*.jpl"`

The following sections explain each type in detail.

Displaying Screens

A control string can open a client screen as a form, as a stacked window, or as a sibling window. Control strings that open screens have the following syntax:

```
*[leadChar] [(viewportArgs)] screenName
```

If you omit `leadChar`, Panther opens the screen as a form. A single ampersand (`&`) opens the screen as a stacked window, while a double ampersand (`&&`) opens it as a sibling window.

Refer to Chapter 13, “Developing Client Screens,” and Chapter 24, “Setting the Screen Sequence,” for more information about how Panther manages screens as forms and windows.

Search Path

Panther looks for the named screen in the following places in this order:

- The memory-resident screen list.
- All open libraries.

If all searches fail, Panther displays an error message and returns.

Viewport Arguments

You can optionally specify arguments for the screen's viewport—that is, the window in which the screen is displayed. Viewport arguments determine the screen's position on the physical display, the viewport's dimensions, and the offset of the screen's contents within its viewport as follows:

```
(row, col, len, width, vRow, vCol)
```

Note: All viewport arguments are optional. However, if you specify any one argument, you must supply all leading arguments; trailing arguments are optional.

`row, col`

The position of the viewport's top left corner on the physical display, where *`row`* and *`col`* are one-based offsets from the physical display's top left corner. The physical display excludes any area already used either by the screen manager, such as a base window border, or by the application's menu bar. Thus, arguments of 1,1 start the screen at the first line and leftmost column that are available.

Signed integer values (negative or positive) specify the viewport's position relative to the previously active screen. For example, the following control string invokes newWindow.scr as a sibling window whose viewport is two rows higher and two rows left of the current screen's viewport:

```
&&(-2,-2)newWindow.scr
```

If the window does not fit on the display at the specified location, Panther adjusts it as needed. Panther does not allow viewports to be positioned completely offscreen.

`len, width`

The viewport's dimensions in rows and columns. A value of 0 or less specifies to use the screen's actual dimensions if the physical display is large enough. Note that the border is counted as part of the screen.

`vRow, vCol`

The row and column of the screen to display on the viewport's first row and column.

If you specify *`vRow`* or *`vCol`*, the cursor appears in the upper-left corner of the viewport, whether or not a field is there. The cursor responds to the cursor keys until it encounters an unprotected field, or the `TAB` key is pressed. When it is in a field, the cursor uses the normal tabbing order among fields.

Table 18-2 contains several examples of control strings that open screens.

Table 18-2 Control strings that open screens

Control string	Action
`mainmenu`	Open `mainmenu` as a form at the physical display's top left corner.

Table 18-2 Control strings that open screens

Control string	Action
`&(5,20)custInfo`	Open `custInfo` as a stacked window at row 5, column 20 of the physical display.
`&&(1,1,10,40,5,5)detail`	Open detail as a sibling window in a 10 row x 40 column viewport at the physical display's top left corner. Row 5, column 5 of the screen is initially displayed at the top left corner of the viewport.

Executing Functions

A control string that executes a function has the following syntax:

```
^ [(targetString  [; targetString] )] funcName [(arglist)]
```

`funcName`

The name of an installed or built-in function or a JPL procedure or module. An installed function can be one of Panther's library functions or your own. For information about function installation, refer to page 44-5, "Installing Functions." Built-in functions are preinstalled in Panther and begin with the prefix `jm_`. For descriptions of built-in functions, refer to Table 18-4 on page 18-7 or Chapter 3, "Built-in Control Functions," in *Programming Guide*.

Panther looks first among the installed functions for *`funcName`*, then among the JPL procedures modules. For detailed information about this search algorithm, refer to page 19-24, "Precedence of Called Objects."

The function must return an integer. If the integer corresponds to the value of a Panther logical key, then that key is processed. For example, if a function returns PF4, then Panther behaves as if PF4 had been pressed by the user. The function should return 0 if there is no key to process.

`arglist`

One or more arguments to pass to parameters in *`funcName`*. Arguments for installed functions—Panther library functions and your own—must be enclosed in parentheses and delimited by commas or spaces. Arguments supplied to the built-in function `jm_keys` should not be enclosed in parentheses.

`targetString`

The control string can optionally test the return value against one or more semicolon-delimited target strings. Each target string has this syntax:

`[`*`testValue`*` =] `*`controlString`*

Panther compares *`funcName`*'s return value to each *`testValue`*, reading from left to right. If it finds a match, it processes the specified control string. If you omit a test value, Panther processes the control string unconditionally. The control string can itself contain a JPL call with its own target strings; you can thereby nest multiple control strings with recursive calls.

For example, given this control string:

```
^(-1=^(^jm_exit)cleanup; 1=&welcome_scr)process
```

Panther processes the string as follows:

1. Calls the JPL module or procedure `process`.
2. Evaluates the return value from `process` to determine its next action:
   - If `process` returns `-1`, Panther executes `cleanup`. When `cleanup` returns, the built-in function `jm_exit` is called.
   - If `process` returns 1, Panther opens the `welcome_scr` screen.

Table 18-3 shows several control strings that call functions:

Table 18-3 Control strings that call functions

Control string	Action
`^verify(name,idnum)`	Execute the user-written function `verify`, passing variables `name` and `idnum` as arguments.
`^sm_cl_unprot()`	Execute the library function `sm_cl_unprot`.
`^jm_exit`	Execute the built-in function `jm_exit`.

Using Built-in Functions

Table 18-4 lists a summary of the built-in functions. For detailed information on each function, refer to Chapter 3, “Built-in Control Functions,” in *Programming Guide*.

Table 18-4 Built-in function summary

Function	Description
`jm_exit`	Close active screen and return to previous screen.
`jm_gotop`	Return to the top level screen.
`jm_goform`	Open a window that prompts for the name of a screen to display.
`jm_keys` *`logicalKey/strings`*	Place the specified Panther logical keys on the keyboard input queue, to be processed by Panther as if each logical key were pressed in order.
`jm_system`	Open a window that prompts for a program to be executed by the operating system.
`jm_winsize`	Allow user to interactively resize a screen.

Note: The control string `jm_keys XMIT` when attached to a push button causes an infinite loop, This occurs because the act of pressing `XMIT` actually activates a push button.

Invoking Operating System Commands

A control string that starts with an exclamation point (!) temporarily passes control to the operating system. At runtime, Panther passes the string after the exclamation point to the operating system for execution. After program execution is complete, control returns to the application.

Note: In character-mode, Panther displays a message that the user must acknowledge before control returns to the application.

If you include variables in the control string, they must be prefixed by a colon(:). Panther's colon preprocessor expands colon-prefixed variables to their literal values before passing the string to the operating system.

Table 18-5 shows several operating system control strings:

Table 18-5 Control strings that call system commands

Control String	Action
`!ls`	Display a directory listing.
`!vi "newdoc"`	Invoke vi to edit `newdoc`.
`!rm :rmData`	Remove the file whose name matches the contents of variable `rmData`.

CHAPTER

19 Programming in JPL

JPL is an interpreted language with a C-like syntax. Because you can write and edit JPL code within the editor, you can write and execute procedures without interrupting your development work flow. You can also write JPL procedures directly to a library and call them via event-type properties provided in the Properties window for screens and widgets. Use JPL for rapid prototyping and later rewrite the procedures in C. Or leave the code unchanged; JPL can get most jobs done quickly and efficiently.

JPL Modules and Procedures

JPL modules contain one or more procedures written in JPL. You create modules through Panther's own JPL editor (described on page 19-14 under "Writing JPL in the Editor") or in a text editor. Screen modules are created through the Focus⟶JPL Procedures property; widget modules through the Validation⟶JPL Validation property; and report modules through the Inclusions⟶JPL Procedures property. Widget, screen and report JPL modules are saved in the screen binaries; Panther automatically reads these at the appropriate stage of program execution. You can also create library modules directly in the editor workspace. For faster access, you can install library modules in the application's memory-resident list.

Module Structure

A module contains one or more procedures. The first procedure of a module can be unnamed. All subsequent procedures are named through JPL's `proc` command. For example, the following module has two procedures; the first is unnamed, the second named `warning`:

```
if actual_cost > forecast_cost
    call warning()

proc warning()
msg emsg "Value exceeds budget forecast."
```

Unnamed and named procedures are different in two ways:

- A module's named procedures must be called explicitly, while a module's unnamed procedure is called automatically at specific events of program execution. Read "Calls" on page 19-19 to learn how JPL calls named and unnamed procedures.
- Variables that you declare in the unnamed procedure are visible to all procedures in the same module, while variables in a named procedure are visible only to that procedure. You typically declare variables in an unnamed procedure in order to initialize them and make them accessible to all named procedures in the same module.

Refer to Figure 19-1 on page 19-7 to view a sample JPL module.

Parameters

The `proc` command can specify parameters that receive arguments passed by the procedure's caller. You specify parameters as a comma-delimited argument list within parentheses. The procedure's caller can pass in constants, global constants, variables, or colon-expanded variables as arguments. 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 variables in its caller; it can only alter its own copies.

For example, the earlier `warning` procedure is modified below; it now expects its caller to supply two arguments that are copied to `actual` and `forecast`. A more informative message is produced by using the colon-expanded values of these variables:

```
if actual_cost > forecast_cost
   call warning(actual_cost, forecast_cost)

proc warning (actual, forecast)

vars diff = actual - forecast

msg emsg "Value in :actual exceeds budget forecast by $:diff"
```

Passing Standard Arguments to JPL Procedures

If a procedure is called as an event function for a widget or screen—for example, as a screen's exit function—and the function name omits parentheses, Panther automatically passes standard arguments to the procedure. These arguments indicate the current status of the widget or screen. Their number and type vary; for example, two arguments are passed for screens, four for widgets, and three for grid widgets. The procedure's `proc` statement must contain the appropriate number of parameters in order to receive these arguments.

For example, you might define the following procedure in a screen's JPL module in order to handle grid data:

```
proc gridProc( basefld, occ_no, status )
```

You can set `gridProc` in any of several grid properties. When Panther calls this procedure at runtime, it sets parameters `basefld`, `occ_no` and `status` with the three standard arguments associated with grids. So, if a grid's Row Entry Function (`row_entry_func`) property contains the string `gridProc`, Panther calls the procedure each time the cursor enters a new row and sets its three parameters to the grid's base widget number, the number of the current occurrence, and an integer bitmask that describes why the procedure was called.

Note: Precedence is always given to arguments that are specified in the property string. For example, if a grid widget's Entry Function property contains the string `gridEntry(val)`, Panther supplies the contents of `val` to procedure `gridEntry`. If arguments are explicitly omitted through empty parentheses (), Panther does not supply the standard arguments.

The unnamed procedure of the module for a screen or widget module is always supplied standard arguments that indicate the current status of the screen or widget. To receive these arguments, the unnamed procedure must have a `parms` statement.

For more information about the standard arguments available for screens and widgets, refer to page 19-21, “Calls from Screens and Widgets.” The `parms` command description shows how to declare parameters in an unnamed procedure.

Return Types

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. For example, the following sequence of statements passes data from variables `data1` and `data2` to procedure `process_input`, which is defined to return a double precision value. This return value is used to determine whether the `if` statement evaluates to true or false:

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

double proc process_input(d1, d2)
vars retval
//process d1 and d2 values
return retval
```

Procedure Execution

Procedure execution begins with the first statement of the procedure and continues to the end of the procedure, or until a `return` statement executes. If an execution error occurs, Panther aborts execution of the current procedure, posts an error message, and returns to the procedure's caller. In all cases, a procedure returns to its caller when execution ends.

Panther interprets each physical line as a separate statement, unless the line ends with the backslash (\) continuation character. JPL physical lines can be up to 253 characters in length.

Control Flow Statements

By default, Panther executes JPL procedures sequentially from start to finish. You can use JPL's `if`, `else`, `for`, `while`, `switch`, `case`, `default` , `break`, and `next` statements to manipulate the order of statement execution. JPL has no limit to how many levels deep you can nest control flow statements.

Conditional and loop statements (`if`, `else`, `for`, `while`, `switch`) allow curly braces `{ }` as blocking characters so you can conditionally execute multiple statements. Each blocking character must have its own line except to specify a null statement—`{}`. If you nest multiple blocks, make sure that all block characters are paired correctly.

The following example shows an if statement that contains a block of two statements:

```
if cost > 1000
{
    exceptions = exceptions + 1
    msg emsg "The cost is very great."
}
```

A left and right brace on the same line indicate a null statement. In the following example, the `for` statement keeps count while testing a condition. Because no other statements are required, the `for` block consists of a null statement:

```
for i = 1 while str(i, 1) != " "
  { }
```

Included Modules

Panther procedures (including the unnamed procedure) can contain `include` statements that specify a JPL library module. At runtime, Panther compiles and inserts this module within the calling procedure before execution begins.

Include statements have the following syntax:

`include` *`module`*

where *`module`* is any JPL library module. The included module can also contain its own `include` statements. You can nest up to eight include statements.

Panther looks for *`module`* in this order:

1. Memory-resident modules.
2. Library module in all open libraries.

Comments

You can enter commented text in JPL in three ways:

- Prefix the commented string with two slash (`//`) characters. JPL treats all remaining text on that line as a comment.

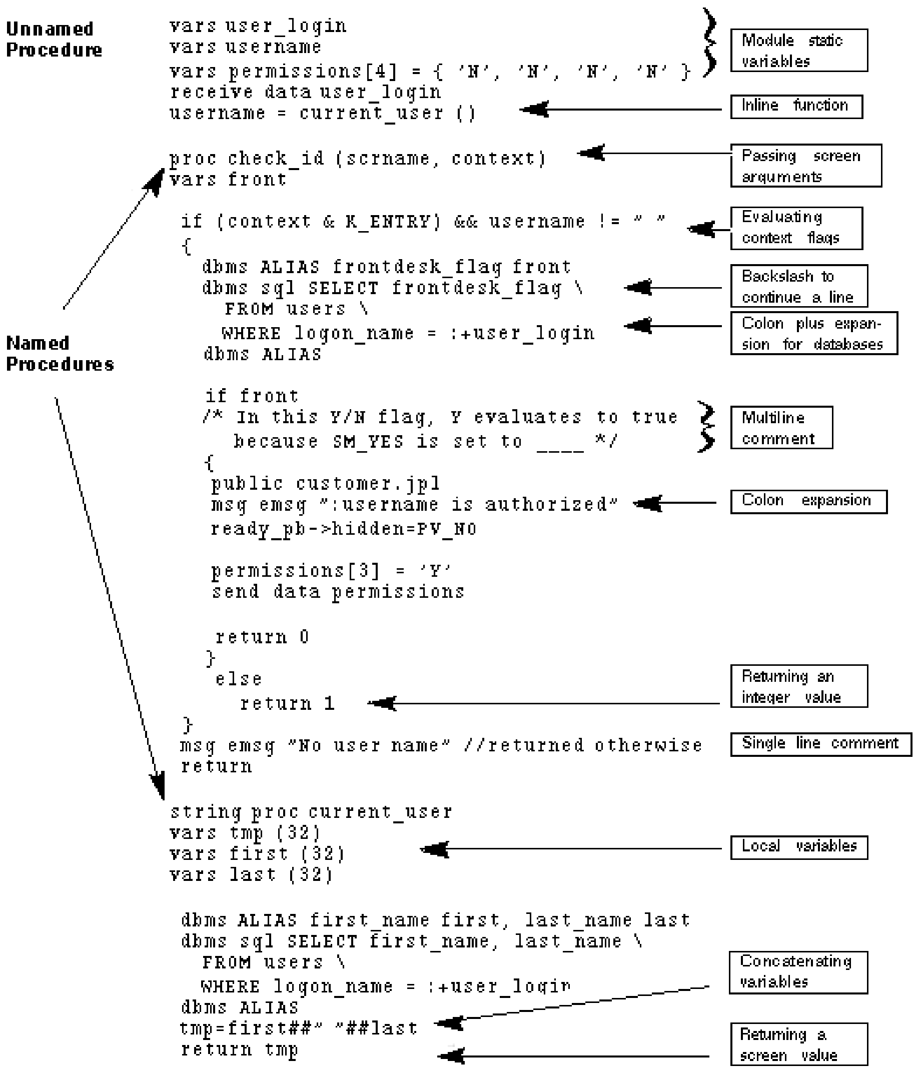
- Enclose the commented string with the block specifiers `/*` and `*/`. JPL treats all text within this block as a comment. Use `/* */` comment characters to comment contiguous lines of text.

  Note: You cannot embed comments within a line of code; all text on a line that follows a comment character, including `*/`, is ignored at runtime.

- Begin the line with a pound (#) character. JPL treats the entire line as a comment.

  Note: The # character must be the first non-blank character of a line in order to be interpreted as a comment character. If it is embedded within a line, JPL interprets it as a reference to a widget by number.

Sample JPL Module



```
vars user_login
vars username
vars permissions[4] = { 'N', 'N', 'N', 'N' }
receive data user_login
username = current_user ()

proc check_id (scrname, context)
vars front

 if (context & K_ENTRY) && username != " "
 {
   dbms ALIAS frontdesk_flag front
   dbms sql SELECT frontdesk_flag \
     FROM users \
     WHERE logon_name = :+user_login
   dbms ALIAS

   if front
   /* In this Y/N flag, Y evaluates to true
      because SM_YES is set to ____ */
   {
    public customer.jpl
    msg emsg ":username is authorized"
    ready_pb->hidden=PV_NO

    permissions[3] = 'Y'
    send data permissions

    return 0
   }
    else
      return 1
 }
 msg emsg "No user name" //returned otherwise
 return

string proc current_user
vars tmp (32)
vars first (32)
vars last (32)

 dbms ALIAS first_name first, last_name last
 dbms sql SELECT first_name, last_name \
   FROM users \
   WHERE logon_name = :+user_login
 dbms ALIAS
 tmp=first##" "##last
 return tmp
```

Figure 19-1 This sample screen-level JPL module makes a JPL module public and displays additional options for front desk employees.

Module Types

Panther lets you create the following types of JPL modules:

- Widget, screen, and report modules that you create through the editor by way of their respective JPL properties. These are saved along with their screen or report binaries.
- Library modules that you can create using the editor or any text editor outside the editor workspace (and then include it in a library). You can also extract library modules, convert them to data structures, and install them in the application's memory-resident list.
- File modules that you can create using any text editor. If desired, these modules can be compiled with `jpl2bin` to be in a binary format.

An application's ability to access the procedures in a JPL module depends on its type and how it is loaded and called. For instance, Panther executes a widget module only during widget validation. The procedures in this module can only be called by each other and are invisible to the rest of the program. Conversely, named procedures in screen modules are available to the entire application while the screen is active.

Panther's ability to access library and memory-resident modules depends on how they are loaded and called. If you load a module into memory as a *public* module, its named procedures are visible to the entire application and can be called directly. If a module is not public, the library in which it resides must be open and the module can only be called by its filename; this invokes the module's unnamed procedure. The named procedures in this module are accessible only through its unnamed procedure.

The following sections describe each module type and how Panther executes it.

Widget Modules

Widget modules are associated with individual widgets. You create and modify widget modules through the widget's JPL Validation property. This property is available for most widget types, including grids and groups. When you select this property, the JPL

Program Text dialog box opens. You use the dialog box's editing window to enter and modify JPL code. For more information on using this editing window, refer to page 19-14, “Writing JPL in the Editor.”

Panther executes a widget module only when it performs validation for the widget. In the case of data entry widgets such as text widgets, validation occurs when the user exits via TAB. For push buttons, radio buttons, check boxes, list boxes, and toggle buttons, validation occurs when the widget is clicked with the mouse or otherwise activated, for example, by the NL key. Because a widget module is accessible only to its widget, use it to perform tasks that are specific only to that widget.

The first procedure of a widget module must be unnamed. The unnamed procedure in a widget module is always this module's entry point. The module can also include named procedures; however, these can only be called by other procedures in the same module. When you save the module, the editor automatically compiles it. If an error prevents compilation, Panther issues a message and returns you to the JPL Program Text dialog box, where you must correct the error.

Because Panther saves the module as part of the widget, you can view and edit this module only through the editor. When you copy this widget to another screen or to the repository, Panther copies the module along with other widget data.

Executing Widget Modules

Panther calls a widget module after it executes the widget's validation function, if one exists. Panther first executes the module's unnamed procedure and passes the standard arguments associated with widget processing. For widgets such as single line text widgets, four arguments are passed that describe the widget and its current status: its field number, contents, occurrence number, and a set of context-sensitive flags. The unnamed procedure must have a `parms` statement in order to receive these arguments. For more information about arguments for different widget types, refer to page 19-21, “Calls from Screens and Widgets.”

Screen Modules

Screen modules are associated with specific screens. All the named procedures in a screen module are available to the application while the screen remains active. You create and modify screen modules through the screen's JPL Procedures property. When you select this property, the JPL Program Text dialog box opens. You use this dialog box's editing window to enter and modify JPL code. For more information on using this editing window, refer to page 19-14, “Writing JPL in the Editor.”

The first procedure of a screen module can be unnamed; an unnamed procedure is optional. All subsequent procedures must be named. When you save the module, the editor automatically compiles it. If an error prevents compilation, Panther issues a message and returns you to the JPL Program Text dialog box, where you can correct the error.

Because Panther saves the module as part of the screen, you can view and edit this module only through the editor. If you save the screen as another file or as a repository entry, Panther copies the module along with all other screen data.

Executing Screen Modules

When you open a screen at runtime, Panther loads all its named procedures into memory. It then executes the screen module's unnamed procedure, if any. Panther passes the two standard arguments associated with screen processing to this procedure: the name of the screen and a set of context-specific flags. The unnamed procedure must have a `parms` statement in order to receive these arguments. For more information about these arguments, refer to page 19-21, “Calls from Screens and Widgets.”

While the screen is active—that is, displayed on top—every named procedure in its JPL module can be called. You can use these procedures to perform any task required by the screen.

Panther executes the unnamed procedure only when the screen first opens, after which it executes the global screen function and the screen's entry function, if any. Panther does not execute a screen module's unnamed procedure on subsequent exposures of an already open screen—for example, when a child or sibling screen closes.

Report Modules

Report modules are similar to screen modules; each report module is associated with a report and saved with the report binary. All named procedures in a report module are available while the report is running.

Unlike screen modules, all report procedures should be named and can be accessed by a corresponding call node in the report structure or called as a subroutine by another process.

You create and modify report modules through the report's JPL Procedures property. When you select this property, the JPL Program Text window opens. For more information on using this editing window, refer to page 19-14, “Writing JPL in the Editor.” When you save the module, the editor automatically compiles it. If an error prevents compilation, Panther issues a message and returns you to the JPL Program Text window, where you can correct the error.

External Modules

External modules are modules saved to disk in libraries or in a file and are therefore not saved or associated with any particular Panther screen. You can extract library modules and install them in the application's memory-resident list. Unlike widget and screen modules, external modules are available to the entire application at any time. Panther finds the modules in memory, in open libraries, or on disk. For details on the search order for library modules, refer to page 19-24, “Precedence of Called Objects.”

External modules are accessible to the application in two ways:

- Call the module by name. Panther executes its first, unnamed procedure. If a module is not loaded through the `public` command, its library must be open or it must be findable along the path set by the `SMPATH` variable and you must call the module by name. An unloaded module has only one entry point, its unnamed procedure.
- Call the named procedures in a public module, that is, a module loaded through JPL's `public` command. When Panther loads a public module, it loads the module's procedures into memory and executes its unnamed procedure, if any. The application can call any named procedure in a public module until it is removed from memory through the `unload` command.

Library Modules

You can create library JPL modules from within the editor (choose File→New→JPL) or with any text editor (and add them to a library). You can also write the contents of widget and screen JPL modules to a library, thereby making them accessible to other screens and to the application as a whole if necessary.

File Modules

File modules can be stored in ASCII, in binary, or in an ASCII/binary format. Modules that are stored as ASCII files are easy to modify and are available to the entire application. However, because Panther must recompile the module each time it is called, an ASCII file module also incurs more processing time than screen or widget modules. To improve performance, precompile the module with `jpl2bin`. If an error occurs during compilation, Panther issues an error message and returns to the module's caller.

Module Compilation

Panther compiles library modules when you save them from within the editor. The module is saved in binary format and Panther uses this compiled format at runtime. If the compiler finds syntax errors, it issues a warning and lets you save the module in an uncompiled format.

The source file is also stored to the library. Library module names should conform to operating system conventions. For filtering purposes, use a `.jpl` extension on library modules that you create within the editor.

You can call a library module only if its library is already open (via `sm_l_close` or on startup via the `SMFLIBS` variable). Panther loads the module into memory each time you call it.

If you create the JPL module outside of the Panther environment, you can later store the module in a Panther library. To store a disk file module in an application library, you can perform either of the following procedures:

From the editor:

1. Be sure a library is open (choose File→Open→Library).
2. Choose File→New→JPL. An untitled JPL dialog box opens.
3. Choose Edit→Read File. The Insert JPL File dialog box opens.
4. Select the desired JPL file and choose OK. The text is inserted at the cursor position in the JPL editing window.
5. Choose File→Save. The Save JPL Module dialog box opens, where you can choose the library in which to store the module.

From the library table of contents:

1. Compile the module with the `jpl2bin` utility from the command line (on page A-20).
2. Be sure a library is open (choose File⟶Open⟶Library).
3. Open the Library Table of Contents dialog box (choose View⟶Library TOC).
4. Choose Add...
5. Select the desired JPL file and choose OK.

From the command line:

1. Compile the module with the `jpl2bin` utility (on page A-20).
2. Add the module to the library using the `formlib` utility (on page A-14).

Memory-Resident Modules

You can add a JPL module to an application's memory-resident list. Making a JPL module memory-resident reduces I/O time. The module is held in memory during the life of the application; therefore, ample memory might be required to run your application.

You add a module to the memory-resident list in these steps:

1. Extract the module from its library with `formlib`.
2. Compile the module with `jpl2bin`.
3. Convert the binary file to source with `bin2c`.
4. Install the array with the function `sm_formlist`.

You must recompile your application after creating or editing a memory-resident list. For more information on memory-resident lists, refer to page 42-8, “Including Memory-Resident Components.”

Writing JPL in the Editor

JPL modules are created within the editor:

- Widget and screen modules are accessed through their JPL Validation and JPL Procedures properties, respectively.
- Report modules are accessed via their JPL Procedures property.
- Library modules can be created and accessed via the File menu.

The JPL modules created through these properties are saved with the screen binary in an ASCII and binary format. The ASCII version allows you to view and edit the JPL; the binary version is used at runtime.

Screen- and Report-Level JPL

Selection of the screen and report JPL property invokes the JPL Program Text window, where you can examine and edit the JPL code currently stored with that property:

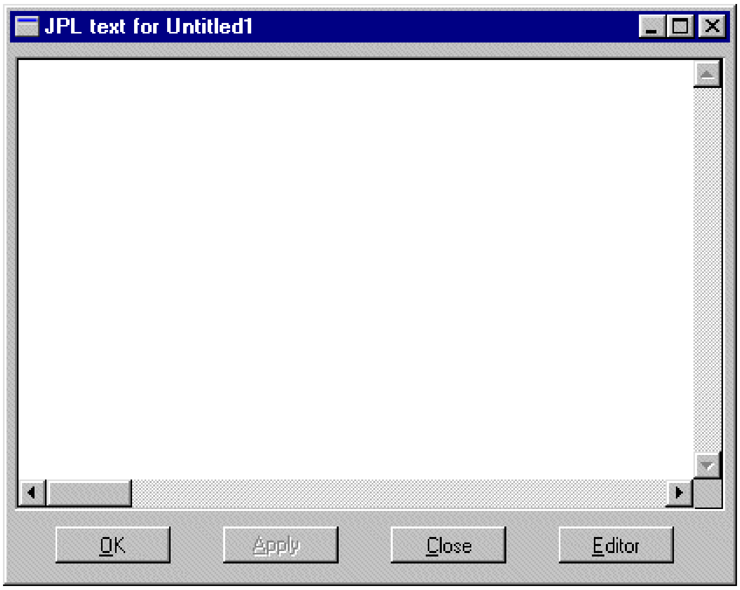


Figure 19-2 Create or edit screen-and report-level JPL in the JPL Program Text dialog box.

Screen- and report-level JPL is compiled and saved with the screen binary. Therefore, if the JPL compiler detects a syntax error, you must correct the error before you can save the module. However, you can save it to disk by choosing File⟶Save As⟶ASCII Text File.

When writing or editing screen-and report-level JPL, you can take full advantage of the editor's File and Edit menu options.

To access the default text editor, choose Editor, or select the Direct to External Editor option to automatically open the window using the default text editor.

Widget-Level JPL

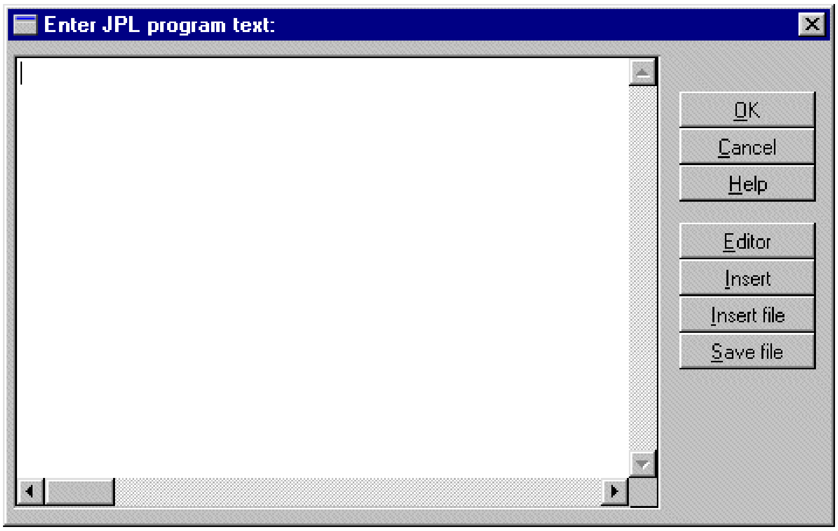


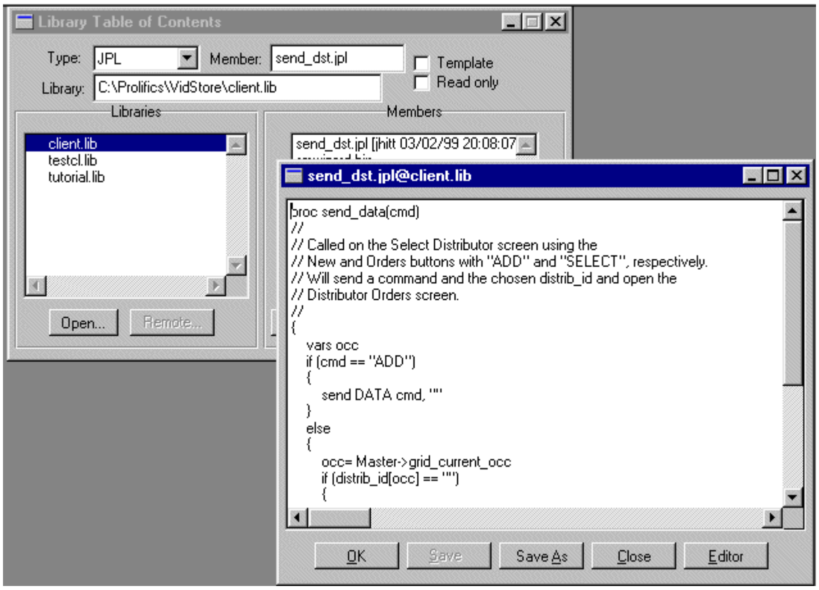
Figure 19-3 Create or edit widget-level JPL code in the JPL Program Text dialog box.

Widget-level JPL is compiled and saved with the screen binary. Therefore, if the JPL compiler detects a syntax error, you must correct the error before you can save the module. However, you can save it to disk by choosing the Save File button.

Since the widget-level JPL dialog box is modal, you cannot access the editor workspace menus.

Library Modules

You can create and edit library JPL modules within the editor by opening a library and then choosing File→New→JPL of File→Open→JPL to access an existing module. The JPL text dialog box opens.



```
proc send_data(cmd)
//
// Called on the Select Distributor screen using the
// New and Orders buttons with "ADD" and "SELECT", respectively.
// Will send a command and the chosen distrib_id and open the
// Distributor Orders screen.
//
{
   vars occ
   if (cmd == "ADD")
   {
      send DATA cmd, ""
   }
   else
   {
      occ= Master->grid_current_occ
      if (distrib_id[occ] == "")
      {
```

Figure 19-4 Library JPL modules can be created and updated from within the editor.

The name of the module and the library in which it resides are displayed in the title bar in the form: `module@library_name`.

Use the Edit menu options to copy, cut, and paste text, use File→Save options to save the module to an open library, or use File→Save As→ASCII Text File to save the module as a disk file.

The JPL is compiled when you save the module to a library. If the compiler detects a syntax error, a warning is issued and you can choose to correct the error or save the module in an uncompiled format (Save As→ASCII Text File) which you can collect later.

Using Your Own Editor

You can type your JPL directly into the dialog box, or you can invoke your local editor, for example, Notepad in Windows or vi in UNIX, by choosing Edit→External Editor (for widget-level JPL, choose the Editor button in the JPL Program Text window). If the Options→Direct to External Editor option is selected, your preferred text editor is invoked immediately on entry into the JPL dialog box. The local editor is defined by the configuration variable `SMEDITOR`. When you exit the editor, you are returned to the JPL dialog box, which contains your latest edits.

You can also use your favorite text editor outside of the Panther workspace to write your JPL modules. Later you can compile the modules and import them to the appropriate library.

Note: If you exceed the maximum line length of 253 characters, Panther issues an error message when you try to return to the dialog box and returns you to your editor to make the necessary corrections.

Inserting JPL To and From Disk

You can write and read code to and from disk files as well. To read a disk file into the JPL window, choose Edit→Read File (for widget-level JPL, choose the Insert File button). To save a JPL module to disk, choose File→Save As→ASCII Text File (for widget-level, choose the Save File button. These invoke the Insert JPL Text File and save JPL Text File dialog boxes, respectively. When you read a disk file into a module, Panther inserts its contents at the cursor's current position.

To insert JPL from another library, choose Edit→Insert From Library (for widget-level, choose the Insert button).

The dialog box accepts line lengths of up to 253 characters. If you try to read from a file that contains longer lines, Panther copies all text preceding the erroneous line into the editing window, then issues an error message.

Compiling and Saving

To compile and save a JPL module to its originating library, choose File→Save As→Library Member (for widget-level, choose the Insert button). If an error prevents compilation of a library module, a message is issued and you can correct the error or save the module in an uncompiled format which you can correct later.

To compile and save screen- and report-level JPL, choose File→Close→JPL (for widget-level, choose the OK button). If an error prevents compilation while compiling screen_ or widget-level JPL, the editor issues a message and returns you to the JPL dialog box.

Note: Library modules that are referenced (by an include statement) within a module are not checked for compilation errors.

Calls

An application can call JPL modules and their named procedures from various screen and widget hooks, and from control strings. The same calling options are available for any installed C or built-in function. Unless otherwise indicated, all references to procedures in this section apply equally to JPL procedures and installed C functions. For more information about installing C functions, refer to Chapter 20, "Writing C Functions."

Panther provides several ways of issuing calls:

- Enter the name of the module or procedure to execute in the Properties window of a screen or widget—for example, in a widget's Validation Function property, or a screen's Entry Function property.
- Call a module or procedure from a control string.
- Explicitly call a module or procedure through the `call` command.
- Issue an inline call, where the name of the procedure or module name to call is embedded inside a JPL expression and is evaluated to its return value.

You can also call JPL from C through the library function `sm_jplcall`.

A screen module's named procedures can be called from outside the module while the screen is active. Named procedures in library modules are accessible if the module is public; otherwise, the procedures can be called only by the module's unnamed procedure. Named procedures in a widget module can be called only by the module's unnamed procedure.

Arguments

All calls can supply comma-delimited arguments to their corresponding parameters. Enclose the arguments in parentheses. If the procedure takes no arguments, use the void argument specifier (). You can pass the following as arguments:

- Variables, including those declared by the `vars` command, widget 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.

Panther passes arguments by value, so changes to the receiving parameter's value leave its corresponding caller's argument unchanged. If you call an installed C function, you must prepare it for installation with the correct macro (`SM_INTFNC`, `SM_STRFNC`, `SM_DBLFNC`, or `SM_ZROFNC`) in order to pass arguments by value to that function. Refer to Chapter 20, "Writing C Functions," for more information about installing functions.

Note: If the message file sets `SM_DECIMAL` to a comma (,), insert a space between numeric constant arguments; otherwise, JPL interprets the comma that separates these arguments as a decimal point. Both methods shown in the following examples are equally successful in avoiding this problem:

```
ret = (1, 2)
ret = (3 ,4)
```

Returns

A procedure always returns to its caller with a return value—either integer, string, or double, according to the procedure definition. If the procedure lacks an explicit `return` statement, or the `return` statement omits a return argument, the procedure returns to its caller with a value of 0 or an empty string. If an execution error causes the procedure to return prematurely, it returns with -1.

Calls from Screens and Widgets

You can specify modules and procedures in various properties of screens and widgets—for example, in a screen's Exit Function property. If you call a JPL module or procedure and supply no arguments, Panther automatically passes arguments that describe the state of the calling screen or widget. The called procedure must define the parameters needed to receive these arguments.

You can supply the same arguments to a C function if it is installed appropriately for the object that calls it. For example, if a C function is installed as a screen function, it can be called on screen entry and exit and receive arguments that describe the state of the screen. You can install C functions to be called from a screen, widget, group, or grid. For more information, refer to Chapter 44, "Installed Event Functions."

Table 19-1 describes the properties that can specify calls to a JPL procedure and the default arguments that are passed:

Table 19-1 Default arguments passed to procedure

Caller	Property	Arguments
Screen	Entry Function Exit Function	*`screenName, flag`*
Widget	Entry Function Exit Function Validation Function	*`widgetNum, widgetContents, occurrenceNum, flags`*
Grid widget	Entry Function Exit Function Row Entry Function Row Exit Function Validation Function	*`baseWidgetNum, occurrenceNum, flags`*
Tab card widget	Card Entry Function Card Exit Function Hide Function Expose Function	*`cardObjectId, flags`*
Group	Entry Function Exit Function	*`groupName, flag`*

For example, if a widget's Exit Function property specifies the procedure `fld_xt` and no arguments are specified, Panther automatically passes in four arguments to this procedure; the second of these arguments is the widget's current value. `fld_xt` gets this value in parameter `val` and tests it as follows:

```
proc fld_xt (num, val, occ, flg)
{
if val = 'MR'     sex = 'M'else     sex = 'F'
return
}
```

The flag or flags that Panther passes are bit values, which you manipulate through JPL's bitwise operators `&` (`AND`), `|` (`OR`), and ~ (one's complement). You can test these flags for conditional processing when you use the same procedure to handle different execution stages of a Panther object—for example, entry and exit of a widget. For information on flags that are set:

- For a screen, page 44-10, “Screen Functions”
- For a widget, page 44-14, “Field Functions”
- For a grid widget, page 44-20, “Grid Functions”
- For a group, page 44-25, “Group Functions”

Using a Memory-resident Screen

If a screen is memory-resident, Panther passes a null string to the called procedure instead of the screen's name.

Calls from Control Strings

You can use control strings to call procedures on specific input, for example, keyboard input or menu choices. You issue calls from a control string as follows:

^[(*target-string* [; *target-string*])] *name* [(*arg-spec*)]

where *name* can be the name of a procedure or module, and *arg-spec* is one or more comma- or space-delimited arguments to pass to parameters in *name*. The control string can optionally test the return value against one or more semicolon-delimited target strings. Each target string has this syntax:

[*test-value* =] *control-string*

Panther compares name's return value to each *test-value*, reading from left to right. If it finds a match, it executes the specified control string. If you omit a test value, Panther executes the control string unconditionally. The control string can itself contain a JPL call with its own target strings; you can thereby nest multiple control strings with recursive calls.

For example, given this control string for a push button:

```
^(-1=^(^jm_exit)cleanup; 1=&welcome_scr)process
```

Panther calls the JPL module or procedure `process` when the user chooses this push button. It then evaluates the return value from `process` to determine its next action: either to call `cleanup`, or to invoke the `welcome_scr` screen. On return from cleanup, Panther unconditionally calls the built-in function `jm_exit`.

Refer to Chapter 18, "Programming Control Strings," for more detailed information about control string syntax.

JPL Call Command

You call a JPL procedure or module through the `call` command from other procedures or modules. The `call` command uses this syntax:

```
call executable ( [arg-spec] )
```

where *executable* can be the name of a module or procedure, and *arg-spec* is one or more comma-delimited arguments optionally to pass to parameters in *executable*. The entire argument list is enclosed in parentheses.

Inline Calls

Because JPL evaluates a procedure call to its return value, you can embed a procedure call within any expression. The following statement embeds a call to the `credit_eval` procedure:

```
if credit_eval() == 1
    msg emsg "Creditworthy applicant"
else if credit_eval() == 0
    msg emsg "Reject application"
```

You can also specify a procedure as an argument to another procedure. In the following statement, JPL first calls `foobar`, then passes its return value into `foo` as that procedure's second argument:

```
ret = foo(a, foobar(b), c)
```

Precedence of Called Objects

When Panther processes a call, it cannot know whether the called object is a JPL module, a JPL procedure, or an installed function. Panther attempts to execute a JPL call by searching for functions and JPL modules or procedures in this order:

1. An installed C or built-in function.
2. If the call is issued from a JPL module, a named procedure in that module.
3. A named procedure in the current screen's module.
4. A named procedure in a public module. If the procedure name exists in more than one public module, Panther uses the procedure in the most recently loaded module.
5. A memory-resident module.
6. A library module in an open library.

Variables

JPL recognizes four kinds of variables:

- JPL module variables declared by the `vars`, `proc`, or `parms` commands.
- Global JPL variables declared by the `global` command.
- Screen variables—widgets, groups, and LDB entries.
- Panther variables, which begin with the `@` character.

This chapter shows how to declare and reference variables in JPL.

Declaring JPL Variables

Earlier sections in this chapter showed how JPL declares parameter variables through the `proc` and `parms` commands. You can also declare a JPL variable with the `vars` command. JPL variables are not typed; you can assign a variable any string or numeric value. All values are stored as strings.

The `vars` command declares one or more JPL variables:

```
vars var-spec [, var-spec]
```

`var-spec` specifies the variable's name and properties as follows:

```
var-name [[num-occurs]] [(size)] [= init-value]
```

The following sections describe required and optional elements in a variable declaration.

`var-name`
: The name of the variable, where var-name 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 (.).

`[`*`num-occurs`*`]`
: Optionally declares var-name as an array of *`num-occurs`* 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 `fname` with a size of 15 bytes:

```
vars fname (15)
```

If the value assigned to a variable is too large for its allocated size, Panther truncates it. For example, if `fname` is programmatically assigned a value of `Russell-Carrington`, it accepts only the first 15 characters, `Russell-Carring`.

= *init-value*

Optionally initializes the variable to *init-value*, where *init-value* 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 (`""`).

Declaring Global Variables

You can declare global variables that are recognized throughout the application with the following syntax:

`global` *var-spec [, var-spec]*

where *var-spec* specifies the variable's name and properties as follows:

var-name [[*num-occurs*] *] [* (*size*) *] [* = *init-value*]

Like the `vars` command, `global` can declare multiple comma-delimited variables; each declaration can optionally declare the global as an array, specify its size (1 to 255 bytes), and assign its initial value.

To reinitialize or clear a global variable, declare it again.

Panther Variables

Panther supplies several predefined variables where it stores application status information. These global variables (beginning with the character `@`) are automatically defined at application startup and maintained by Panther.

After each `dbms` statement is executed, one group of global variables contains any error, warning, or status information returned by the database engine. Refer to "Variables for Logging Error and Status Information" on page 37-4 for information on the variables, such as `@dmengerrmsg`, available through the database interface.

In the transaction manager, global variables contain information about transaction manager processing, such as the occurrence being processed. For more information, refer to Chapter 36, “Runtime Transaction Manager Processing.”

When a Web browser makes a request in a Web application, the HTTP header fields are stored in global variables starting with the characters `@cgi`. Refer to Chapter 11, “HTTP Variables,” in the *Web Development Guide* for more information.

The `@NULL` variable can be used for any parameter in a C function that accepts `NULL` as an argument.

Caution: The Panther `@` variables can be updated frequently. If a variable's value is needed for further processing, copy its value to another location.

Variable Scope and Lifetime

JPL's ability to reference a variable depends on the variable's scope and lifetime. LDB entries, widgets, and groups can be referenced by any module. LDB entries are available as long as their LDB remains loaded in memory. Widgets and groups are available as long as their screen is in memory. Global variables are available for the duration of the application.

Variables declared in an unnamed procedure are accessible to all procedures in the module; those declared in a named procedure are known only to that procedure.

Variables declared inside a procedure remain in memory until the procedure returns, while variables declared in the unnamed procedure 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.

Colon Preprocessing

JPL's colon preprocessor expands any colon-prefixed variable to its literal value. This lets you reference variables in any JPL statement whose syntax otherwise excludes variables; for example, you can embed variables in a string. You can also supply JPL variables as arguments for several JPL commands that take only literal values as arguments, for example `dbms` and `public`.

The preprocessor expands colon-prefixed variables to their literal values before JPL executes the statement. For example, you can reference the variable `acctno` in a `msg` command, even though the command takes only a string value. For example:

```
msg emsg "I cannot find account number :acctno."
```

The colon preprocessor expands `:acctno` to its assigned value before execution. Thus, if `acctno` has a value of 91956, Panther executes the statement by displaying this message:

```
I cannot find account number 91956.
```

Conversely, the following statement:

```
msg emsg "I cannot find account number acctno."
```

yields this message:

```
I cannot find account number acctno.
```

Notes: The colon preprocessor always expands a variable to a string value. You can use this in order to force treatment of numeric values as strings.

Syntax

A colon variable begins with a colon and ends with any non-expandable character, such as a blank or newline, as shown in the following syntax:

```
:var-name
```

Panther has two variations of colon variable syntax for applications that use its database interface, `:+` and `:=`. For more information on these, refer to page 30-1, "Colon Preprocessing."

To prevent expansion of variables that contain colons, prefix the colon with another colon (`::`) or backslash (`\:`), or follow it with a space. In the first two cases, the colon preprocessor discards the first colon or the backslash. In the third case, the colon and following space are preserved.

Expansion

After Panther compiles and loads a JPL module, the colon preprocessor scans each statement from right to left for colons. When it finds one, it starts expansion during which it:

1. Checks for a left parenthesis immediately after the colon, then begins to accumulate characters from left to right.
2. If a left parenthesis exists, the preprocessor accumulates characters until it encounters a right parenthesis. Otherwise, it continues until it encounters a character that cannot be expanded, such as space or a quote character.
3. Tries to identify this string as a variable according to the precedence rules described earlier on page 19-24 under “Precedence of Called Objects.”
4. Expands the variable to its current value, then returns control to JPL for statement execution.

Controlling Expansion with Parentheses

Parentheses explicitly delimit three scope of expansion. For example:

```
vars ref x4
vars alpha[3] = {"bits", "centuri", "rays"}

ref = "alpha"
x4 = :(ref)[3]  // Now x4 = rays
```

The colon preprocessor expands `:(ref)` to `alpha`. JPL then assigns the value of `alpha[3]`—`rays`—to the variable `x4`.

Substring Expansion

If a substring specifier immediately follows a variable name, the colon preprocessor gets the specified characters from the expanded value. If you enclose the variable name with parentheses, the colon preprocessor ignores the specifier, and JPL uses the specifier when it executes the statement.

For example, given these variables and assignments:

```
vars xyz = "Belgium"
vars xy = "New Zealand"
vars abc = "xyz"
vars m
```

the following statement assigns the value `New Zealand` to variable `m`:

```
m = :abc(1, 2)
```

The colon preprocessor expands `:abc(1, 2)` to the first two characters of the expanded value that is, it expands `:abc` to `xyz`, then extracts `xy` from that value. After the expansion, JPL assigns to `m` the value of `xy`, which is `New Zealand`.

By contrast, examine the following statement, where the expanded variable is enclosed by parentheses:

```
m = :(abc)(1, 2)
```

This time, the colon preprocessor expands `:abc` to `xyz`. After the expansion, JPL executes the substring specifier on the value of `xyz`—`Belgium`—and assigns its first two characters `Be` to `m`.

For more information, refer to page 19-50, “Substring Specifiers.”

Array Expansion

Colon preprocessing recognizes the subscript, or index, of an array reference as part of the variable and expands it accordingly. If an array reference omits the array's occurrence number, the colon preprocessor concatenates all the non-blank array occurrences and inserts a space between each pair of occurrence values.

The following examples show how Panther expands array references, given these variable declarations and assignments:

```
vars xyz[3] = {"alpha", "beta", "gamma"}
vars alpha[3] = {"bits", "centuri", "rays"}
vars v = "alpha"
vars w = "xyz"
vars x1 x2 x3 x4 x5
x1 = xyz[3] // x1 = gamma
```

1. The colon preprocessor expands `:xyz[1]` to `alpha`. Thus, `:xyz[1][3]` becomes `alpha[3]`. JPL changes the value of `x2` to `rays`:

   ```
   x2 = :xyz[1][3] // x2 = alpha[3] = rays
   ```

2. The colon preprocessor expands `v` to `alpha`. `x3` then equals the third occurrence of `alpha`, which is `rays`. The parentheses enclosing `v` prevent the colon preprocessor from trying to expand the third occurrence of `v`:

   ```
   x3 = :(v)[3] // x3 = alpha [3] = rays
   ```

3. The colon preprocessor tries to replace `:v[3]` with the third occurrence of `v`. Because `v` has only one occurrence, Panther displays an error message:

```
x5 = :v[3] // error occurs because v[3] does not exist
```

4. The colon preprocessor concatenates all non-blank occurrences of `xyz`, separating the occurrences with single blank spaces. `:xyz` must be enclosed in quotes; otherwise, Panther displays an error message because `beta` and `gamma` are not variables:

```
x4 = ":xyz"
// x4 = ":xyz[1] :xyz[2] :xyz[3]" = "alpha beta gamma"
```

Reexpansion

By default, the colon preprocessor evaluates colon-expanded text only once, even if the expanded text itself contains another colon reference. For example, the following code yields display of the message `Thank Goodness, it's :day`:

```
vars day = "Friday"
vars period = "day"
msg emsg "Thank goodness it's :period"
```

To display the message `Thank goodness it's Friday`, append an asterisk (`*`) to the colon:

```
msg emsg "Thank goodness it's :*period"
```

When the colon preprocessor finds a reexpansion operator, it repeats expansion from the rightmost character of the expanded text. You can nest reexpansion operators to reexpand the same text more than once.

Constants

JPL has the following constant types:

- Numeric: an optionally signed sequence of digits with an embedded decimal point. Because JPL performs data type conversions when necessary, you can represent a numeric constant without decimals.

- Date: a literal date enclosed in parentheses. Date constants must use the date format specified in the message file entry `SM_CALC_DATE`. The default in the message file is `%m/%d/%4y`—that is, `MON/DATE/YR4`.
- String: Zero or more characters enclosed by single or double quotation characters.

Non-Decimal Number System Formats

In addition to decimal numeric constants, Panther supports octal, hexadecimal, and binary numeric formats. Panther recognizes these formats through a number's leading characters, shown in Table 19-2:

Table 19-2 Non-decimal numeric formats supported by Panther

Numeric format	Leading characters	Example
Binary	`0b, 0B`	`0b1 + 0b2 = 11`
Octal	`0`	`02 * 04 = 10`
Hexadecimal	`0x, 0X`	`0x3 * 0x5 = E`

If your application requires decimal numbers with leading zeros, you can turn off support for octal numbers by setting the setup variable `OCTAL_SUPPORT` to `OCTAL_SUPPORT_OFF`. The default setting is `OCTAL_SUPPORT_ON`.

Quoted String Constants

String constants are widely used in JPL, especially in msg and invocation statements. At runtime, JPL strips off the quote characters. You can use single or double quote symbols; however, the same symbol must open and close the string constant:

```
"55 Baker St."
'(212) 555-1212'
```

A quoted constant with no characters—`""` or `''` is a null string.

To reference variable values in a string constant, use the colon preprocessor:

```
"The amount is :total"
```

To use a special character in a quoted constant—colon, quote character, or backslash—prefix the character with a backslash.

Setting Properties Using the Property API

Panther objects and their properties can be referenced through JPL. For example, this `if` statement conditionally unhides a widget (`emp_salary`) at runtime by changing its `hidden` property to `PV_NO`:

```
if (login == "super")
   emp_salary->hidden = PV_NO
```

The basic JPL syntax for referencing a Panther object and, optionally, any of its properties is as follows:

```
object-spec[ -> property-spec]
```

The following sections describe syntactical elements and options.

Object Specification

You specify a Panther object either by its name or with object modifiers as follows:

```
object-name
@object-modifier(object-identifier)
```

For example, you can refer to the widget named `last_name` as follows:

```
last_name
@widget("last_name")
```

Object Modifiers

Object modifiers make explicit the type of object required. Panther provides an `@` modifier for each type of Panther object (except JPL variables): `@widget` for widgets, `@screen` for screens, and so on. Use these modifiers to avoid name conflicts—for example, between a screen that is being used simultaneously for data input and as an LDB. They are also useful for referencing objects whose names are otherwise considered illegal—for example, a screen whose name begins with a number. Thus, you can reference a screen with the name `1001.frm` as follows:

```
@screen("1001.frm")
```

Each object modifier takes either a string or integer argument. The argument can be a constant or variable, or an expression that evaluates to a string or integer. Table 19-3 lists the available modifiers and valid arguments for each.

Table 19-3 Object type modifiers in JPL

Modifier	Argument	Examples

`@app`	Always `@app()`. The string identifier `@app()` always refers to the current program and allows access to application-wide properties. Use `@app()` to reference the application directly.	`gui = @app()->in_gui`
		`ms_fld = @app()->mouse_field`

`@id`	An integer handle that uniquely identifies an application object. This integer can be obtained from an object's `id` property or by calling `sm_prop_id`.
	
	Because each object's `id` property is unique, you can use `@id` to reference objects that have the same name—for example, multiple instances of the same screen, or widgets on different screens that have the same name.
		`nextObj = \`
		`sm_list_objects_next(objList)`
		`@id(nextObj)-> mdt = PV_NO`

`@screen`	The name of a Panther screen that is on the window stack. To specify the active window, supply `@current` as a string.	`@screen("custlist.frm")`
		`@screen("@current")`

Table 19-3 Object type modifiers in JPL *(Continued)*

Modifier	Argument	Examples

`@screen_num`	The number of a Panther screen that is on the window stack, where 0 is the active window, -1 is the window below it, and so on.
	Positive numbers number from the bottom of the window stack: 1 is the base window, 2 refer to the window above it, and so on.
		`@screen_num(0)`
		`@screen_num(sm_wcount())`

`@ldb`	The name of an LDB screen.	`@ldb("sales_data.ldb")`

`@widget`	The name of a widget or group on a screen that is on the window stack or in an active LDB. To specify the current widget, supply `@current` as a string.	`@widget("city")`
		`@widget("@current")`

`@field_num`	The number of a widget on a screen that is on the window stack or in an active LDB. Panther consecutively numbers all widgets that accept data from top to bottom and from left to right.
	If a widget has more than one element (`array_size > 1`), `@field_num(element_num)` always evaluates to the first element. Refer to "References to Element Field Numbers" on page 19-37 for more information.
	`@field_num` cannot be used to reference static labels, boxes, lines, graphs, and grid widgets.
		`@field_num(1)`
		`@field_num(numflds - n)`

Table 19-3 Object type modifiers in JPL *(Continued)*

Modifier	Argument	Examples
`@tp_req`	(JetNet/Oracle Tuxedo only) A string identifier (*`callid`*) associated with a service request. The identifier is stored in the `tp_return` application property immediately after a service call is initiated.To specify the current service, supply `@current` as a string.	`@tp_req("@current")->`*`property`*

`@obj`	An object reference to be used in the following functions to specify a method's parameter or a property's value: `sm_obj_call` and `sm_obj_set_property`	`@obj(`*`object-spec`*`)`
		`a=sm_com_load_picture("a.bmp")`
		`ret=sm_com_call_method(images, "Add", 1, '', @obj(a))`

Array Subscripts

If a widget or JPL variable is an array, you can reference occurrences and elements within that array. Occurrences are specified with the following syntax:

```
object-name[n]
@object-modifier(object-identifier)[n]
```

Array elements are specified with the following syntax:

```
object-name[[n]]
@object-modifier(object-identifier)[[n]]
```

The subscripts `[`*`n`*`]` and `[[`*`n`*`]]` indicate the occurrence and element to reference, respectively, where *n* can be a signed (and for occurrences, unsigned) integer constant or expression. Occurrences and elements are both one-based. For example, `@widget("customer")[3]` refers to the third occurrence in `customers`, while `@widget("customer")[[1]]` refers to the array's first element.

If a widget or JPL variable is an array but no occurrence is specified, Panther uses the default occurrence. When executing a field entry, field exit, or validation function, the default occurrence is the occurrence currently being processed. Otherwise, the default occurrence is 1.

Signed and Unsigned Subscripts

Panther interprets an unsigned subscript as an absolute offset within an array. An unsigned subscript must be between `1` and the array's `max_occurrences` property. Because references to an array element must be absolute, subscripts for element references are always unsigned.

Panther treats a signed subscript as a relative offset from the array's current occurrence. You can reference the current occurrence as *`array-name`*`[+0]` or *`array-name`*`[-0]`. For example, the following JPL procedure, called as array `steps`'s exit function, increments by 1 the value in `steps`'s current occurrence and puts it into the next occurrence:

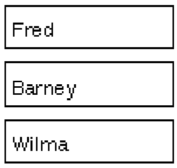
```
proc nextOccData( widget_num, data, occ, context )

if ( occ < steps->max_occurrences )
{
   steps[+1] = steps[+0] + 1
}
return
```

Relative references that use an expression for a subscript must enclose the expression in parentheses. For example `steps[+(n)]` refers to the *n*th occurrence after the current one in array `steps`, while `steps[(+n)]` refers to the array's *n*th occurrence.

References to Element Field Numbers

Although Panther assigns unique field numbers to elements within an array, any reference to these elements through their field number always evaluates to the array itself—that is, its first element. For example, the following array has three elements, which are numbered 1 through 3:



Given this array, the following statement puts `Wilma` into variable `data`:

```
data = @field_num(3)
```

To use field numbering to reference a given element's data, you must translate that field number into an occurrence number. For example:

```
proc getElemData()
vars cur_elem, i, elem_data, occ_no

// get the current element's field number
cur_elem = @screen()->fldnum

// get occurrence number of data in current element
for i = 1 while i <= @field_num(cur_elem)->array_size
{
   if @field_num(cur_elem)[[i]]->fldnum = current_elem
   {
      occ_no = @field_num(cur_elem)->first_occurrence + i - 1
      elem_data = @field_num(cur_elem)[occ_no]
   }
}
```

Precedence of Object Types

If a named object's type is not made explicit, Panther searches for that object among the following Panther types, in this order:

1. Local variables already declared in the current JPL procedure
2. Variables that are global to the current JPL module
3. Widgets or groups on the current screen
4. Widgets in an active LDB (local data block)
5. Global JPL variables
6. Screens in the window stack, starting with the active screen
7. Active LDBs

Compound Object Strings

You can join multiple object strings in a *compound object string* with the `!` character. Compound object strings have this syntax:

`*object-string* !*object-string* [ !*object-string*]...`

For example, the following object string specifies the `customer` widget on the active window:

```
@screen("@current")!@widget("customer")
```

Compound object strings let you make the context of a Panther object as specific as you like and avoid possible ambiguity among different objects with the same name. For example, if two screens on the window stack—`custqry.frm` and `custedit.frm`—both have a `cust_id` widget, you can uniquely identify each one as follows:

```
custqry.frm!cust_id
custedit.frm!cust_id
```

You can achieve even greater specificity within a compound object string by including object modifiers. For example, all of the following object strings are variants of `custqry.frm!cust_id`:

```
custqry.frm!@widget("cust_id")
@screen("custqry.frm")!cust_id
@screen("custqry.frm")!@widget("cust_id")
```

Note: You can reference objects through their object IDs with the `@id` modifier; these unique handles provide a way to identify an object that is independent of its name and context.

Object Values

An object's value is implicit in all references to it. In practice, this applies only to widgets that can have values. For example, you can get and set the contents of a text widget or a push button's label; widgets such as lines and boxes have no equivalent values that you can access.

In the case of arrays, subscripted references return the value of the specified occurrence or element; non-subscripted references return the first element. Thus, these two statements put the same data into variable `cust`:

```
cust = @widget("customer")[[1]]
cust = @widget("customer")
```

You can get portions of an object's value by appending substring specifiers to the object's reference. For example, this statement gets the first eight characters from `customer`'s second occurrence:

```
cust = @widget("customer")[2](1, 8)
```

For more information about substrings, refer to page 19-50, “Substring Specifiers.”

Properties

All Panther objects have properties that can be accessed with this syntax:

```
object-spec ->property-spec
```

The string that you supply for *`property-spec`* contains at a minimum the JPL mnemonic for the desired property. For example, you can reference the current screen's title as follows:

```
@screen("@current")->title
```

If a property can be set to multiple values, *`property-spec`* can specify one of them; for more information, refer to page 19-41, "Multi-item Properties." You can also specify a portion of a string property's setting; this is described on page 19-42 in "Property Substrings."

Property specification can include the `@property` modifier. For example, you can reference the current screen's title as follows

```
@screen("@current")->@property("title")
```

The `@property` modifier is optional if you use the property's actual mnemonic; it is typically used in order to reference a property through a variable. For example, an all-purpose procedure that changes a widget's properties at runtime might look like this:

```
proc change_props (widg_name, prop, value)

@widget(widg_name)->@property(prop) = value
```

Editor Properties

If a property is accessible through the editor, its JPL mnemonic is usually a variant of the name used in the Properties window, where all characters are in lower case, and non-alpha characters such as spaces, dashes, and slashes are replaced with underscores. For example, the Menu Name property is referenced as `menu_name`.

A number of exceptions exist, usually for properties that share the same label in the Properties window. For example, if you set a widget's FG Color Type and BG Color Type properties to Basic, both properties get Color Name as a subproperty. To differentiate these two properties, their runtime names are `fg_color_name` and `bg_color_name`, respectively. For a full list of runtime property names, refer to Chapter 1, "Runtime Properties," in *Quick Reference*.

Note: Several properties that are visible in the Properties window are not accessible at runtime—for example, the Inherit From and Columns properties.

Runtime and Application Properties

Panther also provides access to a number of properties that are not available in the editor, either because they are accessible only at runtime or because they are application-wide (`@app`) properties. For example, `selected` is a runtime property that returns true or false for a specified occurrence in a list box or selection group; `in_gui` is an application property that returns true if the application is running on a GUI platform and false if in character mode.

Refer to Chapter 1, "Runtime Properties," in *Quick Reference* for a list of all properties and their definitions.

Multi-item Properties

Some properties have an array of values, for example, the Drop-Down Data property of combo boxes and option menus. To reference multi-item properties, specify the offset into that property's values as follows:

`object-spec` `->`*`property-name`* `[`*`prop-item`*`]`

For example, the following code changes the selected item in an option menu that has its Drop-Down Source property set to constant data:

```
#replace current item with contents of "substitute"
vars count
for count = 1 \
    while flavors->drop_down_data[count] != flavors
{}
flavors->drop_down_data[count] = substitute
flavors = flavors->drop_down_data[count]
```

To access control string assignments for a screen or for the application, use the desired logical key as the `control_string` property's offset. For example, the following statement gets the screen-level control string assigned to the PF5 key:

```
ctrlstr = @screen("@current")->control_string[PF5]
```

Although properties cannot have properties, you can call `sm_n_num_occurs` and `sm_n_max_occur` for property expressions. For example:

```
drop_down_size = sm_n_num_occurs("optmenu->drop_down_data")
```

Property Substrings

You can get and set a portion of a string property's value with the following syntax:

```
object-spec ->property-name(offset, length)
object-spec ->property-name [prop-item ](offset, length)
```

For example, the following code conditionally assigns the first eight characters of a widget's name to its `column_title` property:

```
if @field_num(i)->column_title = ""
  @field_num(i)->column_title = @field_num(i)->name(1, 8)
```

Property Value Types

Properties can be grouped into three general categories according to the types of values that they take:

Literal Properties

Literal properties take any value—string, integer, or numeric, depending on the property. For example:

```
@widget("customers")->first_occurrence = 1
@screen("@current")->control_string[XMIT] = "^verify_acct"
```

Some properties have implied or explicit ranges. For example, you cannot set an array's `first_occurrence` property to a value greater than the number of occurrences in the array.

Logical Properties

Logical properties take a value of `PV_YES (1)` or `PV_NO (0)`. For example:

```
@widget("salary")->focus_protection = PV_YES
```

Enumerated Properties

Enumerated properties can only be set to one of several predefined integer constants. For example, a widget's hidden property can be set to one of three constants: `PV_YES`, `PV_NO`, or `PV_ALWAYS`.

For a full listing of runtime JPL property names and valid values, refer to Chapter 1, “Runtime Properties,” in *Quick Reference*.

Implicit Properties

All widgets that contain data have a property that let you set its initial value—Initial Text for text widgets, Label for push buttons and check boxes, and so on. For most widget types, these need not be referenced explicitly. To access a widget's data, refer to the widget itself. For example, the following statements change the labels of check boxes `day1` through `day7` to the values found in successive elements of array `lang`:

```
for count = 1 while count <= 7
   @widget("day"##count) = @widget(lang)[count]
```

Table 19-4 shows which widget types are referenced directly in order to change their data, and the editor properties that set their initial data. There are accessible at runtime as implicit properties.

Table 19-4 Editor properties setting a widget's initial data

Property name	Widget types
Initial Text (Format/Display)	single line and multiline text, list box, combo box, option menu
Initial Value (Input)	scale
Label (Identity)	dynamic label, push button, check box, radio button, toggle button

Notes: Graph widget data is set by its Value Source property; this multi-item property must be explicitly referenced, for example, `@widget("sales")->y_value_source[1]`.

Properties of Elements and Occurrences

Properties of an array's occurrences and elements can be accessed by subscripting the array reference—a single pair of square brackets refer to occurrences `[]`, double square brackets to elements `[[]]`. For example, this statement toggles the `reverse` property of an array's first element:

```
salaries[[1]]->reverse = !salaries[[1]]->reverse
```

Selection Group Data

In the editor, you can group together multiple radio buttons, check boxes or toggle buttons into a selection group. JPL identifies a selection group name as an array whose number of occurrences is equal to the number of selections from the group. Each array occurrence contains the number of the selected item: the first element contains the number of the first selected item, the second element contains the number of the next selection, and so on.

Groups can be set up to accept one, multiple, or no selections. If the group allows only one selection—its `num_of_selections` property is set to `PV_0_OR_1` or `PV_1`—its corresponding JPL variable is an array with one occurrence of data, where *`group-name`* `[1]` contains the number of the selected item. Because single-selection groups have only one occurrence, JPL lets you omit the subscript. Thus, *`group-name`*`[1]` and *`group-name`* are equivalent.

For example, `days` is a selection group that allows multiple selections. It contains seven check box widgets with these labels:

```
[ ] MON  [ ] TUE  [ ] WED  [ ] THU  [ ] FRI  [ ] SAT  [ ] SUN
```

Panther numbers widgets in this group in order of their placement on the screen: `MON` has a value of 1, `TUE` a value of 2, and so on. If the user selects `THU` and `SUN`, `days[1]` has a value of 4, while `days[2]` has a value of 7.

You can programmatically evaluate and manipulate the contents of the group array. For example, the following code returns the number of items selected from `days`, then passes each selection to the routine `days_off`:

```
occurs = days->num_occurrences
for count = 1 while count <= occurs
{
  call days_off( days[count] )
}
```

You can programmatically change group selections by setting group array occurrences to the desired values. The following code selects members 6 and 7—`SAT` and `SUN`—in group `days`:

```
days[1] = 6
days[2] = 7
```

You can also use library functions `sm_select` and `sm_deselect` to change group selections. The following code is equivalent to the previous JPL:

```
call sm_select("days", 6)
call sm_select("days", 7)
```

Grid Properties

If you place a widget inside a grid widget, the widget becomes an array whose size is determined by the number of occurrences assigned to the grid widget.

At runtime, the current selection inside of a grid widget can be determined by `grid_current_occ`, a read-only property. The following JPL statement returns the number of the selected row in a grid named `Detail`:

```
myvar=@widget("Detail")->grid_current_occ
```

Traversal Properties

When you use the transaction manager, it builds a tree of all table views that are linked to the root table view. It traverses this tree to issue transaction manager commands to each table view or server view. You can query traversal properties to get information about the table views, server views, and links that are a part of the current transaction.

The following JPL procedure queries the `sv` property to ascertain the server view for the current widget on widget entry. It then executes the `VIEW` command to specify that server view:

```
proc get_sv_query
if K_ENTRY
{
  vars value1
  value1 = name->sv
  call sm_tm_command("VIEW :value1")
}
return 0
```

If the specified property references an object that does not participate in the current transaction, Panther returns an error. For more information on traversal properties, refer to page 36-19, “Using Traversal Properties.”

Global Variables

You can reference any JPL variable declared by the `global` command at any time during the application. JPL also recognizes global variables defined in Panther header files—for example, logical key names such as `XMIT` and `EXIT`, and bit mask settings such as `K_EXPOSE` and `K_ENTRY`. You can reference these variables in any JPL expression and pass them as arguments to another procedure or function.

Caution: Because Panther uses these variables internally, avoid changing their values; doing so can yield unpredictable and possibly harmful results.

Data Types, Operators, and Expressions

Data types describe how JPL uses the values of variables and constants. Operators specify what to do or how to manipulate variables and constants. Expressions combine variables and constants to produce new values.

Data Types

JPL determines the data type of a variable or expression according to its value or usage. All variable values are stored as character strings; JPL converts those values when required.

JPL recognizes four data types:

- String: zero or more characters. Because all variable values are stored as character strings, no conversion is required. Maximum string lengths are system-dependent.
- Integer: a sequence of digits with no decimal point; the value can be signed or unsigned. JPL converts values of this type to integers. If a numeric value contains a decimal point followed by zeros, JPL treats it is an integer.

- Numeric: a sequence of digits, either signed or unsigned, that contains a decimal point. JPL converts values of this type to floating point.
- Logical: a string, integer, or numeric that evaluates to a logical value—that is, either true or false. If a string, it evaluates to true if it starts with the value of message entry `SM_YES`—for example, `y` or `Y`. The string evaluates to false if it starts with any other character. A numeric or integer evaluates to a logical false if it is 0, and a logical true for all other numbers.

Operators

The following sections summarize JPL operators, their operands, and the data type of the value after the operation. Associativity is left to right except for exponentiation, where it is right to left.

String

JPL string operators evaluate to a string. Operands must also be strings.

`()`	substring specifier
`##`	concatenation

Numeric

Evaluate to an integer or float. Operands must be either an integer or float.

`@date`	date calculation
`@length`	string length calculation
`@sum`	array sum
`^`	exponentiation
`/`	division
`*`	multiplication
`+`	addition
`-`	subtraction

Assignment =

Evaluates to numeric or string, according to the operand types. Both operands must be of the same data type.

Relational

Evaluate to true or false; both operands must be of the same data type.

`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to
`==`	equal to
`!=`	not equal to

Logical

Evaluate to true or false; operands must be logical values.

`!`	`NOT` (unary operator)
`&&`	Logical `AND`
`||`	Logical `OR`

Bitwise

Evaluate to integer; operands must be integer types:

~	one's complement
`&`	bitwise `AND`
`|`	bitwise `OR`

Operator Precedence

JPL operators have the following precedence, in decreasing order:

```
() []  @date @length  @sum
##
^
~  !
/  *
+  -
>  >=  <  <=
==  !=
&
&&
|
||
=
```

Conversion of Operands

Some operators require operands of specific data types. If the operand's data type is different, JPL tries to convert it; otherwise an error occurs. In the case of relational and logical operators, JPL checks whether the operand data types are the same; if they are different but compatible—for example, integer and numeric—JPL converts them to one or the other; if they are incompatible, an error occurs.

Table 19-5 shows the data type that JPL uses for operands of compatible data types in relational and logical expressions:

Table 19-5 Data type conversion in relational and logical expressions

Operand type	String	Float	Integer	Logical
String	string	error	error	logical*
Float	error	float	float	logical**
Integer	error	float	integer	logical**
Logical	logical*	logical**	logical**	logical

Table 19-5 Data type conversion in relational and logical expressions

Operand type	String	Float	Integer	Logical

** A string evaluates to a logical true or false if it begins with the value of SM_YES or SM_NO.*
*** A numeric or integer evaluates to a logical true if it is non-zero or or to a logical false if 0.*

Concatenation

Use the concatenation operator ## to join multiple values into a single string. For example, these statements concatenate the string `Blue Moon` into variable `a`.

```
vars a = "Blue "
vars b = "Moon"
a = a##b
```

Substring Specifiers

Substring specifiers let you reference any part of a string that is in a variable or property. Specify a substring with the following syntax:

`obj-name (offset, length)`

`obj-name`
: The name of a JPL variable, widget, or LDB entry, or a property that takes string values.

`offset`
: The offset of the first character of the substring to get from *`obj-name`*, where the first character in *`obj-name`* is 1. A value for *`offset`* is required, and can be an integer or integer expression.

`length`
: An integer expression that evaluates to the substring's length. If *`length`* exceeds the substring's actual length, JPL reads only up to the last byte of data. A value for *`length`* is optional: if no argument is supplied, JPL operates on all characters from offset to the end of the string.

The following examples show some common uses for substring specifiers:

- Extract a country code from an international phone number.

```
if int_phone(1, 3) == "039"
  country = "Italy"
```

- Find the first blank in a string.

```
for i = 1 while string(i, 1) != " "
{ }
```

- Append a zip code extension.

```
zip(6) = "-"##extension
```

Notes: If the message file sets `SM_DECIMAL` to a comma (,), insert a space between numeric constant arguments; otherwise, JPL interprets the comma that separates these arguments as a decimal point. The space can either precede the comma or follow it.

@date

The `@date` operator lets you compare and perform arithmetic on dates. This operator uses a date as its operand—either a widget with a date format, or a date string constant or expression. `@date` converts a date constant to a numeric by counting the number of days between the date constant and January 1, 1753—the standard for date calculations.

For example, if widgets `order-date` and `ship-date` have date edits, you can add 30 days to `order-date`'s value and assign it to `ship-date`:

```
ship-date = @date(order-date) + 30
```

In the next example, `today` is a widget with the current date, and `days` is a variable that gets the number of days between today and 4/1/96:

```
days = @date("4/1/1996") - @date(today)
```

If an operand includes a time value—for example, `02/22/94 10:15`—`@date` ignores the time value and outputs only a date value.

@length

The `@length` operator counts the number of characters in one or more string arguments. You can supply string constants or variables as arguments. You can use a substring specifier on any argument that is a variable.

`@length` counts all characters and embedded blanks. Leading blanks in right-justified widgets and trailing blanks in left-justified widgets are ignored. In quoted string constants, leading blanks are counted but trailing blanks are ignored.

For example, the following statement gets the total number of characters in `fname` and `lname`:

```
vars ln
ln = @length(@widget("fname"), @widget("lname"))
```

@sum

The `@sum` operator calculates the sum of all non-blank occurrences in an array. In the next statement, `quantities` is an array and `total` is a widget that gets the sum of occurrences in `quantities`:

```
total = @sum(quantities)
```

Bitwise Operators

JPL provides three operators for bit manipulation: `AND` (&), `OR` (`|`), and one's complement (~). Bitwise operators let you examine and set the flags that are set on bit masks.

For example, this procedure tests the value of widget status flags `K_ENTRY` and `K_EXIT` to determine whether the widget is being entered or exited:

```
proc field_func (number, data, occ, flags)
if flags & K_ENTRY
  jpl do_process
else if flags & K_EXIT
  jpl do_exit_process
return
```

The next procedure examines the settings of `K_KEYS` to determine which key the user pressed to exit a widget:

```
proc field_func2(num, dat, occ, flags)
if (flags & K_KEYS) == K_NORMAL
  return
else if (flags & K_KEYS) == K_ARROW
  msg emsg "Please use the tab key to move between fields."
return
```

For more information on the flag settings that Panther passes into widget and screen modules and hook functions, refer to “Calls from Screens and Widgets” on page 19-21 and Chapter 44, “Installed Event Functions.”

Expressions

An expression produces a new value by combining constants, variables, and operators. In all statements, Panther's colon preprocessor evaluates colon-expand ed variables. In all expressions, JPL's statement processor replaces variable names with values. Evaluation is generally from left to right; however, you can affect order of evaluation through parentheses.

JPL evaluates an expression as one of four data types: string, numeric, bitwise, or logical. However, it is an error to combine a string assignment with a numeric assignment for a single variable within one expression. The following sections discuss these data types.

String Expressions

A string expression combines one or more quoted string constants or values of string variables. Substring specifiers and ## are string operators. The following examples are all string expressions:

```
'Montreal'
"Processed :i items"
fname##' '##lname
telephone(1, 3)
```

Numeric Expressions

A numeric expression combines variables and numeric constants with one or more of the numeric operators. The following examples are numeric expressions:

```
y + z
@sum(quantities)
@length(fname,lname)
x^y + y * (z^3/4 + 1) - x/2
86
```

If the setup variable DECIMAL_PLACES is set to a number, JPL rounds the value of a numeric expression to that number of decimal places. You can change this with a format specifier to declare the total length and the number of decimal places. Format specifiers have this syntax:

```
%[ t ] [m] [ .n] var-name
```

where *m* and *n* are integer constants or variables. *m* specifies the total number of characters, including leading spaces, sign, digits, and decimal place. If you omit *m*, or *m* is too small to output the variable's value, JPL uses the variable's size. *n* specifies the number of digits after the decimal place. If you omit *n*, JPL uses 2 decimal places.

For example, the following statement assigns 1.667 to i.

```
%6.3 i = 10/6 // rounds value to 1.667
```

t overrides rounding and truncates to the specified number of decimal places, if any. For example, the following statements truncate the values assigned to variables i and n:

```
%t1.2 i = 10/6  // truncates i to 1.66
%t1.0 n = 10/6  // truncates n to 1
```

Notes: Panther uses the sprintf() function to perform rounding. Because this function's behavior is compiler-specific, rounding results for 0.5 decimals can vary among different platforms.

If *var-name* is a widget or LDB entry, you can define its floating point precision by setting Data Formatting (data_formatting) to PV_NUMERIC and setting its Format Type property. At validation, Panther uses this property to format the widget's value.

Bitwise Expressions

A bitwise expression uses variables or constants which have the data type integer, and any of the bitwise operators. The following examples are bitwise expressions:

```
flag1 & flag2
x | mask
```

Logical Expressions

A logical expression uses logical and relational operators to evaluate variables, numeric constants, integer constants, string expressions, numeric expressions, or integer expressions. Operands must be of the same data type; otherwise, JPL tries to convert them according to Table 19-5 on page 19-49. For example, you can compare a numeric literal to a variable or expression only if JPL can evaluate the variable or expression to a numeric. Otherwise, it displays an error message.

The following examples are logical expressions:

```
y
x != 7
(total * (1 + tax)) <= max_value
flag > ~flag
```

In contrast to C, the JPL interpreter always fully evaluates a boolean expression. In the following example, JPL calls `myFunc` even though the expression already evaluates to true:

```
vars a = 1
if ( a || myFunc() )
...
```

JPL Commands

All the JPL commands are fully described in Chapter 2, “JPL Command Reference,” in *Programming Guide*. In general, command arguments can be either variables or strings. String arguments must be enclosed in single or double quotes. To use a variable's value within a string, you can append a colon to the variable (colon preprocessing). For more information, refer to page 19-27, “Colon Preprocessing.”

Optimization

You can improve performance of JPL procedures in several ways:

- Library JPL procedures can be made memory-resident. Convert the binary JPL to a C data structure with `bin2c`, and then add it to Panther's memory-resident list with `sm_formlist`.
- Load a library module into memory as `public`. Panther keeps its procedures in memory. Modules thus loaded incur some memory overhead, but execute more efficiently.
- Execute loops with `for` instead of `while`. For example, this `for` construct executes more efficiently than the `while` construct that follows it:

```
for i = 1 while i < 10
  {
    ...
  }

while i < 10
  {
    ...

    i = i + 1
  }
```

- Prevent expansion of a string that contains colons by appending a space to the colon. Using a space is more efficient than prepending a backslash (`\`) or an extra colon (`:`) because Panther avoids copying the argument to a buffer to remove extra characters.

CHAPTER

20 Writing C Functions

This chapter presents an overview on:

- Writing your own C or C++ functions.
- Linking your C or C++ functions into your Panther executables.
- Calling your C or C++ functions.
- Calling Panther library functions.

Additional information is in Chapter 44, "Installed Event Functions."

Types of C Functions

Your own C code can be written in external files and linked to your Panther clients (`prodev[.exe]`, or `prorun[.exe]`), your Panther application servers, or your Panther Web application servers (`jserver[.exe]`).

There are two types of C functions in Panther:

- Automatic functions for certain Panther events. Panther automatically calls the pre-installed function when that event occurs. For each event type, only one function can be installed at a time (with the exception of time-out functions). For example, an automatic screen function would execute on screen entry and screen exit of every screen.

- Demand functions. You must explicitly call the function from a Panther component through one of its event properties, such as a widget's Exit property, or call the function in a JPL procedure or C function.

Even though there can only be one automatic function for each event type, a Panther object can have both an automatic function and a demand function. For example, when a screen opens, any automatic screen function executes, and the screen can have a demand function specified in the Screen Entry property which also executes.

Using Automatic Functions

Each type of automatic function, such as an automatic screen function or automatic group function, must be:

- Located in a data structure in funclist.c.
- Installed in `funclist.c` with a call to `sm_install` with the appropriate function type.

The following data structure from funclist.c is for an automatic screen function:

```
struct fnc_data autosc_struct = SM_OLDFNC( 0, auto_sfunc);
```

For a list of the `SM_*FNC` macros, refer to page 44-4, “SM_*FNC Macro.”

`funclist.c` also contains the function `sm_do_uinstalls`, which is called internally by Panther at the beginning of program execution. This function should contain calls to `sm_install` for all the types of functions you install. Calls for all of the automatic functions must be added to the `sm_do_uinstalls` function.

For example, placing the following line of code in the `sm_do_uinstalls` function would install the automatic screen function listed above.

```
sm_install ( DFLT_SCREEN_FUNC, &autosc_struct, (int *)0 );
```

For a further explanation of the arguments to this function and a list of function types, refer to Table 44-1 on page 44-6.

Using Demand Functions

Demand functions must also be located in a data structure in `funclist.c`. `funclist.c` provides empty data structures for screen, group, field, tab card, grid, control string, and prototyped functions. By adding your demand function to the appropriate data structure, you will insure that the function is automatically installed. All the data structures provided in funclist.c already have calls to install them into Panther inside the `sm_do_uinstalls` function, that is also in funclist.c.

To determine the appropriate structure, you need to determine whether the function uses standard arguments or non-standard arguments. Any function using non-standard arguments must be installed into the prototyped functions data structure.

For example, if you write a screen function which needs to know that the screen is being opened (rather than closed, since the automatic screen function executes on both screen entry and screen exit), that information can be found in the screen function's standard parameters. Since the function would use the default parameters, you could add the function to the screen data structure and the function would not need to be prototyped. However, the function's parameters would need to match the number and data type of all the standard arguments passed by default to standard screen functions. Also, this function could only be called on screen entry and exit, because only then will the automatic parameters be generated appropriately.

For prototyped functions, you declare the number and type of arguments. Panther supports a function receiving any combination of strings and integers from zero to five arguments, and functions with six integer arguments.

(If you wish to pass floating point values, pass them as strings and then convert appropriately inside your function. The same method of passing the parameter as a string must be used for all non-integer data-types.) Functions return types can be either character string, integer, double or void.

If a function's arguments do not conform to these requirements—for example, there are more than six, or they include an unsupported data type—you can call it indirectly through a wrapper function.

Writing C Functions

To add your own C code to your application, you must:

- Write the C modules. Additional C files should all begin with:

  ```
  #include smdefs.h
  ```

- Create any needed header files, containing prototypes of your functions.
- Modify `funclist.c`.
- Modify the makefile.
- Make the executable.

It is strongly suggested that you copy your Panther distribution and work in the copy of the distribution when adding C code. At a minimum you should always make a backup of the link directory before making changes to any file in that directory. Or, alternatively, work with a copy of the link directory and leave the original link directory unchanged.

For example, to write a simple C function that opens a dialog box to say “Hello World” and link it to Panther, perform the following steps:

1. In the link directory, create the C code in an external file. The sample file, mycode.c, contains:

   ```
   #include "smdefs.h"

   int printhello()
   {
   sm_message_box("Hello World", "My Installed Function",
          SM_MB_OK, "")
   return (0);
   }
   ```

2. In the include directory, create the header file. The sample header file, myapp.h, contains:

   ```
   extern int printhelloPROTO(());
   ```

Notes: `PROTO` is a macro function designed to generate prototypes compatible with both ANSI and pre-ANSI C compilers. If you know your compiler is ANSI-compatible, you could also write this as "`extern int printhello( void );`".

3. In the link directory, edit `funclist.c`. First, at the top of `funclist.c`, list your header file with the other header files. The header files listed in the sample `funclist.c` are:

   ```
   #include "smdefs.h"
   #include "smkeys.h"
   #include "myapp.h"  /* my installed function */
   ```

4. In funclist.c, add your function to one of the Panther C structures. Since this function does not use standard arguments, it is entered as a prototyped function. With this new function, the data structure appears as follows:

   ```
   static struct fnc_data pfuncs[] =
   {
      SM_INTFNC ("pdummy(i)",             pdummy),
      SM_INTFNC ("printhello()",          printhello)
   };
   ```

5. In `funclist.c`, the call to `sm_install` already occurs for prototyped functions, so there is no need to make your own call.

6. In the link directory, modify your makefile to compile and link your code into Panther:

   ```
   SRCMODS = funclist.o mycode.o
   ```

7. Compile and link your application:

   UNIX: make
   Windows: nmake

8. Restart your development executable:

   ```
   prodev
   ```

Additional information on writing and installing C functions is in Chapter 44, “Installed Event Functions.”

Calling C Functions

Once your C routine is linked into Panther, you can use the JPL call command in JPL procedures, such as:

```
call printhello()
```

Or you can call your function from one of the widget or screen event properties, by setting the property value to match your C function name.

```
Entry Function          printhello()
```

Since your function does not take the standard arguments for screen or widget functions, including the ()'s overrides the standard arguments and the function will be called with no arguments.

You can also use C functions in JPL to assign values to variables using the following syntax:

```
x = getsum(1, 3)
```

Calling Panther Library Functions

Panther has an extensive C function library. By default, the entire Panther C library is linked into your development and runtime executables. Most of the C function names start with the prefix `sm_`; some database interface functions start with `dm_`.

A given C function can have several variants. For example, `sm_gofield` has the following variants:

- `sm_n_gofield` — Specify the field name.
- `sm_e_gofield` — Specify the field name and the element number.

- `sm_i_gofield` — Specify the field name and the occurrence number.
- `sm_o_gofield` — Specify the field number and the occurrence number.

In functions having an object name as one of the arguments, the object's name, the object's identifier or the object's property specifier can be entered. For example:

```
sm_n_getfield ("@id(15)")

sm_n_getfield ("@screen("@current")->title_id")

sm_n_intval("field3->length")
```

Refer to Chapter 5, “Library Functions,” in *Programming Guide* for documentation about each Panther C function.

CHAPTER

21 Java Event Handlers and Objects

You can use the Java programming language as an alternative to JPL or C by writing Java event handlers for the screens, service components and widgets in your Panther applications. Panther 5.40 also has enhanced support for working with Java objects.

Support for the Java language has been implemented differently than Panther's C language support. Unlike C functions, you cannot enter the name of a Java method in widget or screen event properties. With Java, you assign a Java class to the Panther object (screen, service component, widget) in the Java Tag property. This Java class will serve as the event handler for the object.

Java Overview

Java is an object-oriented programming language. Unlike C, in which you write functions to manipulate data stored in separate structures, Java objects are a combination of data and methods (functions) that manipulate that data. The data and methods define a class for the objects that can be constructed from that class. A class is not an object, but is a blueprint for an object. At runtime, a *class factory* creates, or instantiates, instances of the object from the class definition.

Classes are arranged in a hierarchy, which allows a subclass to inherit from a parent class, or superclass. A set of classes is called a package.

Using Java in Panther

By default, Java support is enabled in Panther as determined by the behavior variable `JAVA_USE`. When this variable is set to `JAVA_IN_USE`, on startup, Panther dynamically loads a library to initialize Java support. If the library cannot be found, Panther reports an error that Java support is not enabled.

Since the location of the Java library can vary on different platforms, you can override the default location by setting the variable `SMJAVALIBRARY` to the correct location in your environment or in an initialization file.

Writing Java Code

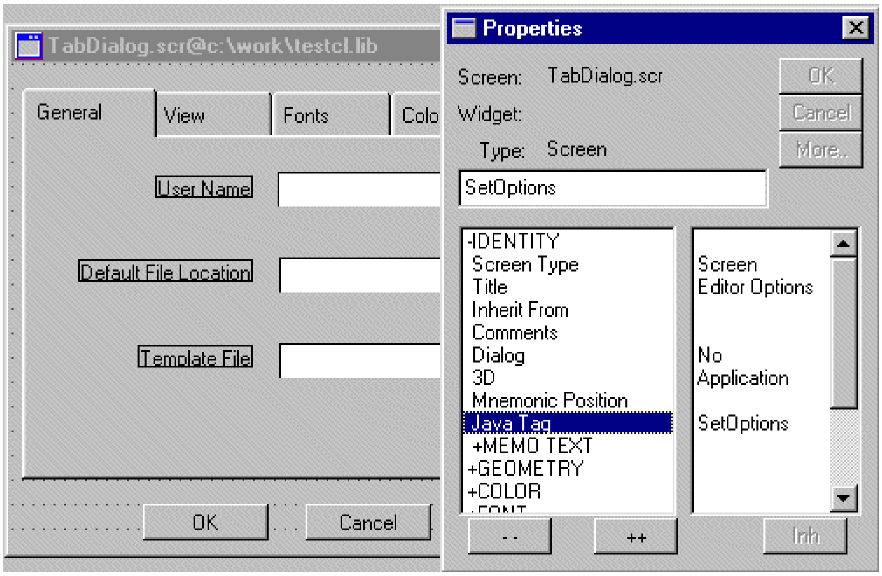
Within the Panther editor, options on the File menu allow you to open, create and save Java files. You can use the text editor specified by `SMEDITOR` or specify a special Java editor with `SMJAVAEDITOR`.

You can compile Java programs in the editor using Tools→Compile Java. The compilation command can be specified by setting `SMJAVACOMPILE`.

Determining the Java Event Handler

Screens, service components and widgets in Panther have a Java Tag property, which can be found under Identity in the Properties window. This property identifies the Java class that is to serve as the event handler for that screen, service component, or widget. At runtime, the Java Tag is passed to a function in the class specified by `SMJAVAFACTORY`, which then provides the event handler itself.

The default `SMJAVAFACTORY` is `ClassTagFactory.java`, which simply assumes that the Java tags are the names of classes. If you want to write another method for matching the Java tags to classes, you can write a your own class factory and specify it using `SMJAVAFACTORY`.



Screen: TabDialog.scr
Widget:
Type: Screen
SetOptions

Screen Type	Screen
Title	Editor Options
Inherit From	
Comments	
Dialog	No
3D	Application
Mnemonic Position	
Java Tag	SetOptions

Figure 21-1 The Java Tag property identifies the Java class that is the event handler.

The event handler classes must provide methods that correspond to the various kinds of events supported by the object (screen, service component or widget) that it is associated with. To this end, predefined interfaces have been provided for the event handler classes to implement. If a class that does not implement the appropriate interface is named as the event handler for a given object, an error will be returned.

At runtime, when an event occurs on an object that has a Java tag set, Panther will send a message to the class that is serving as the object's event handler. The event handler will then perform the action that is programmed for that event. This method will be passed a series of parameters when invoked. Refer to the next section for documentation of each event and a list of the parameters it will receive.

If there is a JPL procedure or C function also associated with the event (in the Properties window), that procedure will be called after the Java event handler has returned. In this way one can associate both methods written in Java and procedures written in C or JPL with the same events, though the Java methods will always be invoked first.

Panther appends the location of its classes (`$SMBASE/config/pro5.jar`) to the `CLASSPATH` environment variable at runtime. As part of your application setup, you must also specify the location of your Java classes in that variable.

Event Handler Interfaces

There are interfaces defined for screens and for the various widget types. In addition to these interface definitions, adapter classes have been provided that provide null implementations of these interfaces. These are located in the distribution at `$SMBASE/config/pro5.jar`. The HTML listing of the interfaces in Javadoc format is at `$SMBASE/config/java`, along with the source class files.

To implement an event handler for a given screen or widget, you would typically subclass the appropriate adapter class, and locally redefine those methods that you wish the screen or widget to support in a non-null manner. For an example, refer to page 21-23, “Java Samples.”

Table 21-1 The interface and adapter class files for widgets and screens

Object	Interface file	Adapter class
ActiveX controls	`ActivexHandler.java`	`ActivexHandlerAdapter.java`
Check boxes	`CheckboxHandler.java`	`CheckboxHandlerAdapter.java`

Table 21-1 The interface and adapter class files for widgets and screens *(Continued)*

Object	Interface file	Adapter class
Combo boxes	`ComboboxHandler.java`	`ComboboxHandlerAdapter.java`
Dynamic labels	`DynamicLabelHandler.java`	`DynamicLabelHandlerAdapter.java`
Grids	`GridHandler.java`	`GridHandlerAdapter.java`
Groups	`GroupHandler.java`	`GroupHandlerAdapter.java`
List boxes	`ListboxHandler.java`	`ListboxHandlerAdapter.java`
Option menus	`OptionmenuHandler.java`	`OptionmenuHandlerAdapter.java`
Push buttons	`ButtonHandler.java`	`ButtonHandlerAdapter.java`
Radio buttons	`RadiobuttonHandler.java`	`RadiobuttonHandlerAdapter.java`
Scales	`ScaleHandler.java`	`ScaleHandlerAdapter.java`
Screens	`ScreenHandler.java`	`ScreenHandlerAdapter.java`
Tab cards	`TabCardHandler.java`	`TabCardHandlerAdapter.java`
Text fields	`TextHandler.java`	`TextHandlerAdapter.java`
Toggle buttons	`TogglebuttonHandler.java`	`TogglebuttonHandlerAdapter.java`

Screen Event Handlers

An event handler for a screen must implement the `ScreenHandler` interface, either explicitly or by subclassing `ScreenHandlerAdapter`. As such it must support the following methods:

```
screenEntry
       void screenEntry(ScreenInterfaces, int context);

screenExit
       void screenExit(ScreenInterfaces, int context);

screenKey
       void screenKey(ScreenInterfaces, int key);
```

```
screenWebEnter
       void screenWebEnter(ScreenInterfaces);
```

```
screenWebExit
       void screenWebExit(ScreenInterfaces);
```

The first parameter passed to each of these methods is a handle to the screen itself. The second parameter passed to the methods `screenEntry` and `screenExit` is an integer bitmask containing the context flags for the event. The second parameter passed to the method `screenKey` is an integer that corresponds to a logical key. This method is used to respond to logical keys, such as the PF-keys (typically pressed by user action) or APP-keys (typically pressed programmatically).

The methods `screenWebEnter` and `screenWebExit` do not receive a second parameter.

ActiveX Control Event Handlers

Event handlers for ActiveX controls only handle the events for the Panther containers used to house the controls. Any events supported by the controls themselves will be implemented by the controls; use `sm_com_set_handler` in the `CFunctionsInterface` to specify event handlers for those events.

An event handler for an ActiveX control must implement the `ActiveXHandler` interface, either explicitly, or by subclassing `ActiveXHandlerAdapter`. As such it must implement the following methods:

```
activexEntry
       int activexEntry(FieldInterface f, int item, int context);
```

```
activexExit
       int activexExit(FieldInterface f, int item, int context);
```

```
activexValidate
       int activexValidate(FieldInterface f, int item, int
       context);
```

Each of these methods receives three parameters. The first is a handle to the ActiveX control itself. The second is an integer containing the control's occurrence number. The third is an integer bitmask containing the context flags for the event.

Check Box Event Handlers

An event handler for a check box must implement the `CheckboxHandler` interface, either explicitly or by subclassing `CheckboxHandlerAdapter`. As such it must support the following methods:

```
checkboxEntry
       int checkboxEntry(FieldInterface f, int item, int context);

checkboxExit
       int checkboxExit(FieldInterface f, int item, int context);

checkboxValidate
       int checkboxValidate(FieldInterface f, int item, int
       context);
```

Each of these methods receives three parameters. The first is a handle to the check box itself. The second is an integer corresponding to the check box's occurrence number. The third is an integer bitmask containing the context flags for the event.

Combo Box Event Handlers

An event handler for a combo box must implement the `ComboboxHandler` interface, either explicitly or by subclassing `ComboboxHandlerAdapter`. As such it must support the following methods:

```
comboboxDoubleClick
       int comboboxDoubleClick(FieldInterface f, int
       item);comboboxEntryint

comboboxEntry
       (FieldInterface f, int item, int context);comboboxExit

int comboboxExit
       (FieldInterface f, int item, int context);

comboboxValidate
       int comboboxValidate(FieldInterface f, int item, int
       context);
```

The first two parameters received by each of these methods are a handle to the combo box itself, and an integer corresponding to the combo box's occurrence number. The method `comboboxDoubleClick` receives only these two parameters; the methods `comboboxEntry`, `comboboxExit` and `comboboxValidate` also receive a third parameter, which is an integer bitmask containing the context flags for the event.

Dynamic Label Event Handlers

An event handler for a dynamic label must implement the `DynamicLabelHandler` interface, either explicitly or by subclassing `DynamicLabelHandlerAdapter`. As such it must support the following methods:

```
labelDoubleclick
      int labelDoubleClick(FieldInterface f, int item);
```

```
labelValidate
      int labelValidate(FieldInterface f, int item, int context);
```

The first two parameters received by each of these methods are a handle to the dynamic label itself and an integer corresponding to the dynamic label's occurrence number. The method `labelDoubleClick` receives only these two parameters; the method `labelValidate` also receives a third parameter, which is an integer bitmask containing the context flags for the event.

Grid Event Handlers

An event handler for a grid widget must implement the `GridHandler` interface, either explicitly or by subclassing `GridHandlerAdapter`. As such it must support the following methods:

```
gridEntry
      int gridEntry(GridInterface g, FieldInterface f, int item,
      int context);
```

```
gridExit
      int gridExit(GridInterface g, FieldInterface f, int item,
      int context);
```

```
gridValidate
      int gridValidate(GridInterface g, FieldInterface f, int
      item,int context);
```

```
gridRowEntry
       int gridRowEntry(GridInterface g, FieldInterface f, int
       item, int context);
```

```
gridRowExit
       int gridRowExit(GridInterface g, FieldInterface f, int item,
       int context);
```

The `GridHandler` methods `gridEntry`, `gridExit` and `gridValidate` are passed four parameters. The first is a handle to the grid itself. The second is a handle to the grid member (field) that currently has focus. The third is an integer corresponding to the grid member's occurrence number. The fourth is an integer bitmask containing the context flags for the event. The methods `gridRowEntry` and `gridRowExit` receive five parameters. The first is a handle to the grid itself. The second is an integer, corresponding to the row number. The third is a handle to the grid member that is gaining or losing focus. The fourth is an integer corresponding to the occurrence number for the grid member that is gaining or losing focus. The fifth is an integer bitmask containing the context flags for the event.

Group Event Handlers

An event handler for a group must implement the `GroupHandler` interface, either explicitly or by subclassing `GroupHandlerAdapter`. As such it must support the following methods:

```
groupEntry
       int groupEntry(GroupInterface g, int context);
```

```
groupExit
       int groupExit(GroupInterface g, int context);
```

```
groupValidate
       int groupValidate(GroupInterface g, int context);
```

Each of these methods receives two parameters. The first is a handle to the group itself. The second is an integer bitmask containing the context flags for the event.

List Box Event Handlers

An event handler for a list box must implement the `ListboxHandler` interface, either explicitly or by subclassing `ListboxHandlerAdapter`. As such it must support the following methods:

```
listboxActivate
       int listboxActivate(FieldInterface f, int item);
```

```
listboxDoubleClick
       int listboxDoubleClick(FieldInterface f, int item);
```

```
listboxEntry
       int listboxEntry(FieldInterface f, int item, int context);
```

```
listboxExit
       int listboxExit(FieldInterface f, int item, int context);
```

```
listboxValidate
       int listboxValidate(FieldInterface f, int item, int
       context);
```

The first two parameters received by each of these methods are a handle to the list box itself and an integer corresponding to the list box's occurrence number. The methods `listboxDoubleClick` and `listboxActivate` receive only these two parameters; the methods `listboxEntry`, `listboxExit` and `listboxValidate` also receive a third parameter, which is an integer bitmask containing the context flags for the event.

Option Menu Event Handlers

An event handler for an option menu must implement the `OptionmenuHandler` interface, either explicitly or by subclassing `OptionmenuHandlerAdapter`. As such it must support the following methods:

```
optionmenuEntry
       int optionmenuEntry(FieldInterface f, int item, int
       context);
```

```
optionmenuExit
       int optionmenuExit(FieldInterface f, int item, int context);
```

```
optionmenuValidate
       int optionmenuExit(FieldInterface f, int item, int context);
```

Each of these methods receives three parameters. The first is a handle to the option menu itself. The second is an integer corresponding to the option menu's occurrence number. The third is an integer bitmask containing the context flags for the event.

Push Button Event Handlers

An event handler for a push button must implement the `ButtonHandler` interface, either explicitly or by subclassing `ButtonHandlerAdapter`. As such it must support the following methods:

```
buttonActivate
       int buttonActivate(FieldInterface f, int item);

buttonEntry
       int buttonEntry(FieldInterface f, int item, int context);

buttonExit
       int buttonExit(FieldInterface f, int item, int context);

buttonValidate
       int buttonValidate(FieldInterface f, int item, int context);
```

The first parameter passed to each of these methods is a handle to the push button itself. The second parameter is an integer corresponding to the occurrence number of the push button. The `buttonActivate` method receives only these two parameters; the methods `buttonEntry`, `buttonExit`, and `buttonValidate` also receive a third parameter that is an integer bitmask containing the context flags for the event. Note that the `buttonActivate` event is invoked when the button is clicked on. It corresponds to the hook accessed in the Properties window by means of the Control String property.

Radio Button Event Handlers

An event handler for a radio button must implement the `RadiobuttonHandler` interface, either explicitly or by subclassing `RadiobuttonHandlerAdapter`. As such it must support the following methods:

```
radiobuttonEntry
       int radiobuttonEntry(FieldInterface f, int item, int
       context);

radiobuttonExit
       int radiobuttonExit(FieldInterface f, int item, int
       context);

radiobuttonValidate
       int radiobuttonValidate(FieldInterface f, int item, int
       context);
```

Each of these methods receives three parameters. The first is a handle to the radio button itself. The second is an integer corresponding to the radio button's occurrence number. The third is an integer bitmask containing the context flags for the event.

Scale Event Handlers

An event handler for a scale must implement the `ScaleHandler` interface, either explicitly or by subclassing `ScaleHandlerAdapter`. As such it must support the following methods:

```
scaleEntry
       int scaleEntry(FieldInterface f, int item, int context);
```

```
scaleExit
       int scaleExit(FieldInterface f, int item, int context);
```

```
scaleValidate
       int scaleValidate(FieldInterface f, int item, int context);
```

Each of these methods receives three parameters. The first is a handle to the scale itself. The second is an integer corresponding to the scale's occurrence number. The third is an integer bitmask containing the context flags for the event.

Tab Card Event Handlers

An event handler for a tab card must implement the `TabCardHandler` interface, either explicitly or by subclassing `TabCardHandlerAdapter`. As such it must support the following methods:

```
cardCardEntry
       void cardCardEntry(FieldInterface f, int context);
```

```
cardCardExit
       void cardCardExit(FieldInterface f, int context);
```

```
cardExpose
       void cardExpose(FieldInterface f, int context);
```

```
cardHide
       void cardHide(FieldInterface f, int context);
```

```
cardTabEntry
       int cardTabEntry(FieldInterface f, int item, int context);
```

```
cardTabExit
       int cardTabExit(FieldInterface f, int item, int context);
```

```
cardValidate
       int cardValidate(FieldInterface f, int item, int context);
```

The first parameter passed to each of these methods is a handle to the tab card itself. The methods `cardTabEntry`, `cardTabExit` and `cardTabValidate` receive as a second parameter an integer corresponding to the tab cards `card_number` property, and as a third parameter an integer bitmask containing the context flags for the event. The methods `cardCardEntry`, `cardCardExit`, `cardExpose` and `cardHide` receive only two arguments, the second being an integer bitmask containing the context flags for the event.

Toggle Button Event Handlers

An event handler for a toggle button must implement the `TogglebuttonHandler` interface, either explicitly or by subclassing `TogglebuttonHandlerAdapter`. As such it must support the following methods:

```
togglebuttonEntry
       int togglebuttonEntry(FieldInterface f, int item, int
       context);
```

```
togglebuttonExit
       int togglebuttonExit(FieldInterface f, int item, int
       context);
```

```
togglebuttonValidate
       int togglebuttonValidate(FieldInterface f, int item, int
       context)
```

Each of these methods receives three parameters. The first is a handle to the toggle button itself. The second is an integer corresponding to the toggle button's occurrence number. The third is an integer bitmask containing the context flags for the event.

Text Field Event Handlers

An event handler for a text field (either single-line or multi-line) must implement the `TextHandler` interface, either explicitly or by subclassing `TextHandlerAdapter`. As such it must support the following methods:

```
textDoubleClick
       int textDoubleClick(FieldInterface f, int item);
```

```
textEntry
       int textEntry(FieldInterface f, int item, int context);
```

```
textExit
       int textExit(FieldInterface f, int item, int context);
```

```
textValidate
       int textValidate(FieldInterface f, int item, int context);
```

The first two parameters passed to each of these methods are a handle to the text field itself, and an integer corresponding to the text field's occurrence number. The method `textDoubleClick` receives only these two parameters; the methods `textEntry`, `textExit` and `textValidate` also receive a third parameter, which is an integer bitmask containing the context flags for the event.

Object Interfaces

The object handles passed to the event handler methods are handles to Java objects that correspond to the Panther objects on the screen. These objects themselves support a variety of methods, and the event handlers that you write will typically make use of these methods.

The interfaces supported by these objects are defined in the files:

```
ScreenInterface.java
FieldInterface.java
GridInterface.java
GroupInterface.java
```

Each of these interfaces extends `WidgetInterface`, which is defined in:

```
WidgetInterface.java
```

Therefore the methods in `WidgetInterface` are common to all the objects that correspond to Panther objects.

The objects corresponding to text fields, push buttons, toggle buttons, check boxes, radio buttons, dynamic labels, tab cards, option menus, combo boxes, listboxes, and scales are all of type `FieldInterface`, and hence support all the methods defined in `FieldInterface.java` and `WidgetInterface.java`.

In addition to the objects that corresponds to the various widgets and screens, there is also an Application object that supports a few methods. The interface for the Application object is defined in `ApplicationInterface.java`.

Implementing Service Component Methods in Java

In general the event handlers for service components are like those of screens. But service components have to respond to a special class of events that the Screen Handler interface doesn't predefine. Service components have to respond to client-initiated requests for methods.

The methods supported by a component are named functions. In COM components or Enterprise JavaBeans, the method names are specified in the editor, in the Component Interface window. In JetNet or Oracle Tuxedo the procedure names are associated with component names in the JIF; each component/procedure pair corresponds to a "service" as far as the middleware is concerned.

At runtime, if a service component has its Java tag property set, the server will attempt to invoke a method with a name corresponding to the name of the method invoked by the client (the name specified either in the Component Interface View or the JIF). Hence if a server component is to implement methods in Java, its event handler must implement methods with names that correspond to the component's public methods. If a Java method with a name corresponding to the method name is found, the Panther server will not continue to look for JPL or C functions with the same name. (This behavior differs from event calls for screen and widget events.) When methods are invoked in this fashion, they will receive two parameters. The first is a handle to the service component itself, this is an object of type `ScreenInterface`. The second is a handle to an interface that supports the middleware-specific functions needed to implement a service.

Service Component Methods in Oracle Tuxedo and JetNet

The methods of JetNet or Oracle Tuxedo service component handlers receive as their second parameter an object that implements `TPFunctionsInterface`. Such an object supports the following methods:

```
int sm_tp_exec(String a1);
```

```
WidgetInterface getTpRequest();
```

```
WidgetInterface getTpRequest(String callid);
```

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.

Service Component Methods in COM/DCOM/MTS

The methods for service components in COM applications receive as their second parameter a handle to an object that implements `ComFunctionsInterface`. Such an object supports the following methods:

```
int receive_args (String text);
```

```
int return_args (String text);
```

```
int raise_exception (int code);
```

```
int log (String text, int code);
```

```
int sm_mts_CreateInstance (String text);
```

```
int sm_mts_SetComplete ();
```

```
int sm_mts_SetAbort ();
```

```
int sm_mts_EnableCommit ();
```

```
int sm_mts_DisableCommit ();
```

```
int sm_mts_IsInTransaction ();
```

```
int sm_mts_IsSecurityEnabled ();
int sm_mts_IsCallerInRole (String role);
```

The functions `receive_args`, `return_args`, `raise_exception` and `log` correspond to the JPL verbs of those names. The rest of the methods correspond to the Panther library functions of the same names.

Service Component Methods in WebSphere

The methods of a service component and Enterprise JavaBean deployed in WebSphere Application Server receive as their second parameter an object that implements `WSFunctionsInterface`. Such an object supports the following methods:

```
public PantherSessionBean get_bean();
```

```
public int log (String message);
```

```
public void raise_exception (String message);
```

```
public int receive_args (String args);
```

```
public int return_args (String args);
```

These methods correspond to the JPL verbs with the same names.

Working with Java Objects

Instantiating Java Objects

Starting with Panther 5.40, there is an alternative to using Java event handlers to integrate Java with your Panther application. Perhaps you want to embed a J2SE class, such as a `HashMap`, into your JPL code. You can do this by making use of the Panther Component subsystem. Although there aren't any non-EJB Java-based Panther Components, the client API for working with Panther Components allows it to work with ordinary Java classes. To do this you set the `current_component_system` property to `PV_SERVER_JAVA` before calling `sm_obj_create`. If your application also uses COM or EJB support, you can change this property as needed.

To access a Java class, you must create an object of the class by calling `sm_obj_create`. This function takes one or more parameters. The first parameter is a string containing the fully qualified class name. This string may also contain a type-specifier that describes the types of any arguments that are needed by the constructor if the constructor is overloaded. If there are arguments needed for the constructor, they are supplies as additional parameters to the `sm_obj_create` call. For example, this is JPL code to create a `HashMap` using the default no-arg constructor:

```
vars hashmap_id = sm_obj_create("java.util.HashMap")
```

Another example, using the `HashMap` constructor that takes an `int` for the initial capacity:

```
vars hashmap_id = sm_obj_create("java.util.HashMap(int)", 100)
```

Type-Specifiers and Arguments

The names passed to `sm_obj_create` and `sm_obj_call` can indicate the types of the parameters to ensure that the correct method is used. The syntax is:

```
<classname-or-methodname>(type[, type] ...)
```

Type-specifiers may follow the class name passed to `sm_obj_create` and the method name passed to `sm_obj_call`. When used, they are included in the class name or method name. Type-specifiers do not conform to Java conventions, but have a custom syntax. The type-specifier is a comma separated list of types, enclosed in parentheses. White space is ignored between the class name or method name and the type-specifier, and between tokens within the parentheses. Types may be primitive or fully qualified class names. There is an exception. For classes in the `java.lang` package, such as `String`, the package name may be left off. The primitive type names supported are: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`.

Arrays may be used in type-specifiers. To indicate an array, the type name, primitive or class name, must be followed by square brackets containing the size of the array. For example `int[5]` represents an `int` array of size 5. The argument corresponding to this type must, therefore, be an array field or JPL array variable containing at least 5 occurrences. Multidimensional arrays are not supported.

When a class name is used as a type, the argument may be a Panther constant, variable, or field name only if the value of the constant, or content of the field or variable can be converted into a Java object of the specified type. For example, Panther will automatically convert a primitive into its wrapper type (for example, `int` to java.lang.Integer) and a Panther value will be converted into a java.lang.String if needed. For other types, use the @obj() syntax to refer to an existing Java object by its Panther object ID. For example,

```
    vars newid = sm_obj_create( \
              "java.util.HashMap(java.util.Map)", @obj(hashmap_id))
```

The return value from sm_obj_create is a Panther Object ID on success, or PR_E_OBJID or PR_E_OBJECT on error. A Java global reference is created for the instantiated Object, so that the instance is not garbage collected by the JVM until sm_obj_delete_id is called for the Object ID.

The use of the Panther Object ID for a Java Object is limited to Panther's sm_obj_xxx family of functions for accessing the object's properties and methods.

Destroying Java Objects

After creating and working with the methods and properties of a Java Object, you should destroy it by calling sm_obj_delete_id. This function takes one parameter, the object ID for the object to destroy. If you don't call this function, the instance will continue to exist until the application terminates, even if the application goes from test to edit mode. In other words, the object will not be garbage collected by the JVM until sm_obj_delete_id is called.

For example, sm_obj_delete_id can be called to delete the HashMap created in the previous code as follows:

```
    call sm_obj_delete_id(hashmap_id)
```

Calling Java Object Methods

To access a Java object's methods, you need to call the function sm_obj_call. The syntax in JPL is:

```
ret = sm_obj_call (objid, methodName(OptionalTypeSpecifier) \
                   [, parm] ...)
```

The first parameter to sm_obj_call is the Panther object ID of the Java object whose method you wish to use. The second parameter is the name of the method you are calling. A type-specifier may be used within this name. The type-specifier is needed if the method is overloaded. The remaining parameters are a comma-separated list of the parameters to the method itself. Field names, variable names, constants, and `@obj()` may be used, exactly as described above in the section Instantiating Java Objects.

If a type-specifier is not used, Panther will try to locate a method in the class with the provided name and number of arguments and will attempt to convert the `parm` arguments to the parameter types needed for the method that was located.

If a method called by `sm_obj_call` returns a `String` or a value that has a primitive type, Panther will return that value. If the method returns a Java object, a Panther object ID will be returned by `sm_obj_call`. Such object IDs can be used in the same way as those returned by `sm_obj_create`. They can be passed in as the first argument of calls to `sm_obj_call`, and must be deleted by calling `sm_obj_delete_id` when they are no longer needed.

Below is JPL code demonstrating the concepts discussed so far:

```
// Initialize the component system to for Java
@app()->current_component_system = PV_SERVER_JAVA

vars a = 1234567891

/* Create a BigInteger using the constructor that takes a
 * String argument. Since JPL can treat the content of
 * 'a' as a String if it is used as such, there is no
 * potential loss of preceision.
 */
vars bigInteger1 = sm_obj_create("java.math.BigInteger(String)",a)

/* Call the 'multiply' method.In this case the 'pow'
 * method could have been used instead. If 'multiply' were
 * overloaded, we could use multiply(java.math.BigInteger)
 * as the second argument below.
 */
vars bigInteger2 = sm_obj_call(bigInteger1, \
                                "multiply", @obj(bigInteger1))

// Convert the result to a string.
vars b = sm_obj_call(bigInteger2, "toString")

// Delete the Object IDs to allow JVM garbage collection.
call sm_obj_delete_id(bigInteger1)
call sm_obj_delete_id(bigInteger2)

msg emsg b           // Displays 1524157877488187881
```

Accessing Panther Functions From a Java Method

The class `com.prolifics.jni.Application` contains the static method `getInstance`. This method returns an `ApplicationInterface` for the Application Object. Using this instance, one can call `getCFunctions`, `getDMFunctions`, `getRWFunctions` and `getTMFunctions`. For example, here is Java code showing this technique:

```
package com.prolifics.samples;
import com.prolifics.jni.*
import java.util.*;
import javax.naming.*;
import javax.naming.directory.*;

public class LdapAuthentication
{
    DirContext ctx = null;

    public int authenticate() {
        int ret = -1;
        ApplicationInterface ai = Application.getInstance();
        CFunctionsInterface   cf = ai.getCFunctions();

        String ldapURL  = c.sm_n_fptr("ldap_url");
        String username = c.sm_n_fptr("username");
        String password  = c.sm_n_fptr("password");

        try {
            Hashtable env = new Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                    "com.sun.jndi.ldap.LdapCtxFactory");
            env.put(Context.PROVIDER_URL,         ldapURL);
            env.put(Context.SECURITY_PRINCIPAL,   username);
            env.put(Context.SECURITY_CREDENTIALS, password);
            ctx = new InitialDirContext(env);
            ret = 0;
        } catch (Exception e) {
            c.sm_message_box(e.toString(),
                         "Authentication Failure", 0, "");
        }
        return ret;
    }
}
```

The above Java code above can be used from JPL as follows:

```
vars ldapAuth = sm_obj_create(
                  "com.prolifics.samples.LdapAuthentication")
```

```
    vars ret = sm_obj_call(ldapAuth, "authenticate")
    call sm_obj_delete_id(ldapAuth)
    if (ret == 0)
        msg emsg "Login Successful"
```

The code shown above is not the best integration of Java with JPL, nor does it handle exceptions very well. It is designed only for demonstrating the techniques discussed in this section.

Panther Library Functions are generally not thread safe. Therefore, avoid creating threads within Java code with the potential to invoke Panther library functions simultaneously from different threads.

Accessing Java Object Properties

Properties of Java objects are class member variables, also called fields. It is recommended with Java Beans that you use the `sm_obj_call` function with setter and getter methods. If this is not an option, you can use the `sm_obj_set_property` and `sm_obj_get_property` functions for this purpose. The following example sets a property on the Java object associated with the id variable:

```
    ret = sm_obj_set_property(id, "PropName", "PropSetting")
```

A type-specifier cannot be used with `sm_obj_set_property`, and the property type may only be a primitive, a primitive wrapper type, or a String. Arrays are not supported. Also, `@obj()` is not supported for the third argument of `sm_obj_set_property`, nor can `sm_obj_get_property` return an object ID.

Designating an Error Handler

You can define an error handler for Java method invocations. For example:

```
    call sm_obj_onerror("ErrorHandlerName")
```

The string passed to `sm_obj_onerror` is the name of the function that you want to use as the error handler. This may be the name of a JPL function, a C prototyped function, or a C control function. If a Java operation (method call, property access, or object creation) fails for some reason, including if a Java method throws an exception, the error handler function will be called. This function is passed three parameters: the property value of the `current_component_system` property - for Java it is `PV_SERVER_JAVA`, the message number, in decimal format, of a message from the

Panther message file for the error, and the corresponding message describing the error. In the case of a C control function, the three parameters are concatenated into one string parameter. The handler does not receive the Java exception string nor the Java stack trace. It only receives a message that there was an unhandled Java exception. It may, therefore, be advisable to cache Java exceptions in a Java object that has methods for retrieving the exception string and stack trace.

Java Samples

The Java methods in these samples are taken from a Panther-built calculator program. The calculator contains a screen event handler for the calculator window and several classes of button event handlers for the push buttons which perform the calculator's operations. The calculator is located in the distribution at `$SMBASE/samples/javacalc`; refer to Appendix C, "Panther Java Calculator," for development notes.

The screen event handler initializes global JPL variables and reprograms some keys. The Java Tag for the screen is `CalcScreen`.

```
import com.prolifics.jni.*;

// This class provides the Screen Entry and Screen Exit
// functions for the Calculator.
// It initializes JPL globals and re-programs
// the BS,DELE and ENTER keys.

public class CalcScreen
   extends ScreenHandlerAdapter
{
   public void screenEntry(ScreenInterface si, int context)
   {
      CFunctionsInterface c = si.getCFunctions();

// Set JPL global variables
      c.sm_n_putfield("op_just_done","1");
      c.sm_n_putfield("operation","");
      c.sm_n_putfield("memory","0");
      c.sm_n_putfield("register","0");
```

```
            c.sm_n_putfield("degrees","0");
            c.sm_n_putfield("currency","0");

// Set <Enter> to generate '=' Key press,
// Backspace to generate "<",
// and Delete to Generate "E"(clear Entry)
            c.sm_keyoption(Keys.NL,   KEY_XLATE, '=');
            c.sm_keyoption(Keys.BKSP, KEY_XLATE, '<');
            c.sm_keyoption(Keys.DELE, KEY_XLATE, 'E');
    }

    public void screenExit(ScreenInterface si, int context
    {
            CFunctionsInterface c = si.getCFunctions();

// Restore default functions of Backspace, Delete and Enter
            c.sm_keyoption(Keys.NL,   KEY_XLATE, Keys.NL);
            c.sm_keyoption(Keys.BKSP, KEY_XLATE, Keys.BKSP);
            c.sm_keyoption(Keys.DELE, KEY_XLATE, Keys.DELE);
    }
}
```

One of the button event handlers is for the push buttons that implement the memory functions of the calculator. The push buttons using this event handler have a Java Tag of `CalcMem`.

```
import com.prolifics.jni.*;

// This class provides A Handler for all calculator
// operations which implement the memory operations.
// There is an if-else-if block which implements
// the individual functions based on the button labels.

public class CalcMem
    extends ButtonHandlerAdapter
{
    public int buttonActivate(FieldInterface fi,int item)
    {
        ScreenInterface si = fi.getScreen();
        CFunctionsInterface c = si.getCFunctions();

        // Save Handle to display since it is used a few times
        FieldInterface fiDisplay = si.getField("display");
        // Get value of Display widget and Key Pressed
        String  key = fi.getfield();
        double dDisplay = fiDisplay.dblval();
        // Get values from JPL global vars
        double mem = c.sm_n_dblval("memory");
        if (key.equalsIgnoreCase("MC"))
            c.sm_n_putfield("memory","0");

        else if (key.equalsIgnoreCase("MR"))
            {
            fiDisplay.putfield(String.valueOf(mem));
            fiDisplay.fval();
            }

        else if (key.equalsIgnoreCase("M+"))
            {
            mem += dDisplay;
            c.sm_n_putfield("memory",String.valueOf(mem));
            }
        c.sm_n_putfield("op_just_done","1");
        return 0;
    }
}
```

CHAPTER

22 Using XML Data

As of Panther 5, you can import and export data from a Panther screen in XML format.

Defining XML Properties

To use the XML feature available as of Panther 5, the screen and its widgets must have the appropriate XML properties set. There are four XML properties that can be specified for widgets or screens:

XML Tag
: A tag associated with the screen, container, or widget.

XML Attributes
: Additional information to be output in the opening tag following the XML Tag value.

  For example, if the XML Tag is `gadget` and the XML Attributes is `magnetic color='blue'`, the opening tag in the generated XML would be:

```
<gadget magnetic color='blue'>
```

Note: If you use special characters, such as > or <, they must be escaped.

XML Prefix
: Data to be output before the opening tag. This property can contain more than one line of text and will be output even if the widget does not have the XML Tag property set.

XML Postfix

Data to be output after the closing tag. This property can contain more than one line of text and will be output even if the widget does not have the XML Tag property set.

Defining XML Screen Properties

The screen being used for XML generation must either have the XML Tag property specified or *both* the XML Prefix and the XML Postfix properties specified.

For screens, if the XML Prefix property is empty, the generated XML starts with the following line:

```
<?xml version='1.0'?>
```

If a different opening line is needed, specify it in the screen’s XML Prefix property.

Processing XML Properties

For screens and other containers (boxes, tab decks, tab cards and grids), Panther generates the following:

- If specified, the XML Prefix property value is output.
- The opening tag and any attributes are output.
- XML is generated for the widgets within the container.
- The closing tag is output.
- If specified, the XML Postfix property value is output.

Note: Even if the container does not have any XML properties specified, widgets in it will be included in the generated XML when they have XML properties set.

For data entry and selection widgets (dynamic labels, single line text widgets, multiline text widgets, option menus, combo boxes and list boxes), Panther generates the following :

- If specified, the XML Prefix property value is output.
- The opening tag and any attributes are output.
- The data from the widget is output.
- The closing tag is output.
- If specified, the XML Postfix property value is output.

Processing XML for Multiple Occurrences

When a grid is converted to XML, the maximum number of occurrences of each grid member having an XML Tag property determines the number of occurrences to output. All offscreen data will be written to the XML.

Processing Hidden Widgets

It is recommended that all widgets used in the XML generation have the same value for Hidden Always. Either all of the widgets must be set to Hidden Always, or none of the widgets must be set to Hidden Always.

Generating XML

You can generate XML by calling one of the following functions:

- `sm_xml_export` and its variants generate the XML to a buffer. Since the buffer can contain TAB and NL characters, `sm_ww_write` should be called put the output in a word wrapped array when calling these functions from JPL.
- `sm_xml_export_file` and its variants write the generated XML to a file.

The form of the function depends on how you specify the screen or LDB.

- To generate XML for the current screen, call `sm_xml_export` or `sm_xml_export_file`.
- To specify the screen by GSD, call `sm_n_xml_export` or `sm_n_xml_export_file`.
- To specify the screen by Object ID, call `sm_obj_xml_export` or `sm_obj_xml_export_file`.

The following JPL procedure generates XML for the current screen to the file `titles.xml`:

```
proc export
   call sm_xml_export_file("titles.xml")
   return
```

Importing XML

You can import XML by calling one of the following functions:

- `sm_xml_import` and its variants import XML from a buffer. Since the buffer can contain TAB and NL characters, `sm_ww_read`.can be used to provide this buffer when calling these functions from JPL.
- `sm_xml_import_file` and its variants import XML from a file.

The form of the function depends on how you specify the screen or LDB.

- To import XML into the current screen, call `sm_xml_import` or `sm_xml_import_file`.
- To specify the screen by GSD, call `sm_n_xml_import` or `sm_n_xml_import_file`.
- To specify the screen by Object ID, call `sm_obj_xml_import` or `sm_obj_xml_import_file`.

The following JPL procedure reads in XML for the current screen from the file `titles.xml`:

```
proc import
   call sm_xml_import_file("titles.xml")
   return
```

Notes:

- In order to write data values into fields, the names of the fields and their corresponding containers must match the names specified in the XML file.
- Data originally exported from a multi-line text widget must be imported back into a multi-line text widget.

Sample XML File

s

The following file contains the data from one of the `videobiz` database screens:

```
<?xml version='1.0'?>
<titles>
<title_id>2</title_id>
<name>Aliens</name>
<genre>SCFI</genre>
<rating>R</rating>
<release_year>1986</release_year>
<title_id>4</title_id>
<name>All the President&apos;s Men</name>
<genre>DRAM</genre>
<rating>PG</rating>
<release_year>1976</release_year>
<title_id>26</title_id>
<name>Moonstruck</name>
<genre>COM</genre>
<rating>PG</rating>
<release_year>1987</release_year>
</titles>
```

CHAPTER

23 Using Widgets

Panther lets you access and manipulate most widgets at runtime, get and modify their data and properties, ascertain the current selection within a radio button group or list box, and determine whether data has changed or passed validation. Panther differentiates between data entry widgets that can be thus accessed and manipulated, and other widgets that are static in nature, like lines, boxes, and static labels.

In order to access widgets, you must know how widgets are identified; for information on widget naming and array/occurrence numbering, refer to Chapter 14, "Identifying Screen Widgets."

Functions described in this section are documented in the *Programming Guide*; refer to that manual for the syntax and specific behavior of each function.

Changing Widget Display

Widget display is determined by the setting for the Hidden property. By default, this property is set to No. You can change the setting to:

- Yes if you want the ability to display the widget at runtime.
- Always if you want to store data in the widget but never display it.

For tab decks, the `topmost_card` runtime property sets the topmost card in the deck or returns the object id of the card that is topmost in the tab deck.

Controlling Input

You can control what type of input in allowed for the widget by setting the widget's properties in the screen editor and, if necessary, changing those properties at runtime.

Setting Data Entry Formats

The Keystroke Filter property sets the criteria for data entry: the keys that can be used (numbers, letters, or a combination) and the format of the entered data.

For example, the following entry in the Edit Mask subproperty would force the user to enter three digits followed by six letters:

```
ID#\9\9\9-\X\X\X\X\X\X
```

ID# and the hyphen are only display characters. Display characters are stripped before sending the value to a database or copying the value with `sm_getfield`§. For more information, refer to “Edit Masks” on page 14-5 in *Using the Editors*.

The Regular Expression property can also enforce a specific pattern of letters or numbers, and in addition, it can restrict the range of characters available. For example, the following expression defines a code of three digits ranging from 1-5, followed by a hyphen and minimum of three letters, but no more than six.

```
[1-5]\{3\}-[a-zA-Z]\{3,6\}
```

For more information, refer to “Regular Expressions” on page 14-7 in *Using the Editors*.

If you want to suggest a format pattern and have that format pattern saved to the database, you can use the Keystroke Filter property in combination with the Initial Text property. For example, an initial text entry of   - - in combination with a setting of Numeric in the Keystroke Filter property would allow the user to enter groups of numbers separated by hyphens.

Other properties in the Input category determine if entry in the widget is required and if data in the widget is protected. For more information, refer to Chapter 14, “Data Entry Widgets,” in *Using the Editors*.

Setting Date and Currency Formats

Under Format/Display, properties determine if a widget's contents are right or left justified, a password, a date format, or a currency format. You can also specify the date or currency format which will be used to enter data. For a description of properties in the Format/Display category, refer to Chapter 10, “Controlling the Way Things Look,” in *Using the Editors*.

Traversing Widgets

You can traverse widgets on the screen using the mouse or the `TAB` key.

Traversing Sets of Widgets

Panther provides a set of library functions that enable you to traverse the contents of a container widgets. These are screens (including ones used as `LDB`s), grid widgets, box widgets, selection groups, synchronized scrolling groups, tab decks, tab cards and table views. Follow these steps:

1. Obtain the container object's id property through JPL or by calling `sm_prop_id`.
2. Call `sm_list_objects_start` to create a list of all widgets that are currently contained by the specified object. This function returns a handle to the list so you can access its contents.
3. reverse the list of objects created by `sm_list_objects_start` through repeated calls to `sm_list_objects_next`. This function, when first called, on a given list returns the object ID of the first widget in the list; each subsequent call returns the object ID of the next widget in the list.

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 within a for loop.

4. Call `sm_list_objects_end` to destroy the object contents list and deallocate the memory associated with it.

For example, the following procedure 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
```

The following example hides a box widget and the widgets positioned within its borders:

```
proc hide_box (name)
{
    vars box_list, count, i, item_id
    @widget(name)->hidden = PV_YES
    box_list = sm_list_objects_start ( sm_prop_id (name) )
    if box_list >0
    {
    count = sm_list_objects_count ( box_list )
    for i = 1 while i <= count
        {
            item_id = sm_list_objects_next ( box_list )
            // don't try to hide static labels
            if @id(item_id)->widget_type != PV_STATIC_LABEL
            {
                // don't try to hide always-hidden widgets
                if @id(item_id)->hidden != PV_ALWAYS
                {
                    @id(item_id)->hidden = PV_YES
                }
            }
        }
        call sm_list_objects_end (box_list)
        return 0
    }
    return -1
}
```

In JPL, the `member` property can be used to find the members of a container. The following example cycles through the cards in the specified tab deck:

```
for i = 1 while i <= my_deck->number_of_cards
{
    a = my_deck->member[i]
    @id(a)-> ...

    //some JPL programming

}
```

Getting Widget Data

Panther library functions let you obtain the data in a widget or its occurrences; they also let you ascertain the widget's current property settings.

Getting Widget and Array Data

The following functions copy data from widgets and arrays:

- `sm_getfield` copies data from the specified widget or occurrence to the supplied parameter. Panther strips leading or trailing blanks.

- sm_fptr returns the contents of the specified widget. Panther strips leading or trailing blanks.
- sm_ww_read copies word-wrapped text from a multiline text widget into a string buffer.
- sm_ww_length gets the number of characters in a word wrap widget.
- sm_dblval returns the contents of the specified widget as a real number.
- sm_intval returns the integer value of the data contained in the specified widget, including its sign. All other punctuation characters are ignored.
- sm_lngval returns the contents of the specified widget as a long integer. It recognizes only digit characters and a leading plus or minus sign.

You can also get information about the data in a widget with these functions:

- sm_dlength returns the length of the data in the specified widget or occurrence of a widget. The length includes any data that is shifted offscreen and therefore out of view. The length excludes leading blanks in right-justified widgets, and trailing blanks in left-justified widgets.
- sm_is_no and sm_is_yes compare the first character of the data in the specified widget or occurrence to the first letter of the SM_NO and SM_YES entries in the message file, ignoring case.
- sm_null lets you test whether a widget's value is null or not. This function checks whether a widget's Null Field property is set to Yes; if it is, sm_null gets the widget's null indicator and compares it to the widget's value.

Getting Properties

You can access all widget properties at runtime through JPL or the property functions: sm_prop_get, sm_prop_set, and sm_prop_id. For example, this JPL if statement conditionally unhides a widget at runtime by changing its hidden property to PV_NO:

```
if (login == "super")
  emp_salary ->hidden = PV_NO
```

For more information about getting and setting widget properties in JPL, refer to Chapter 19, “Programming in JPL.”

Changing Widget Data

Writing Data to Widgets

The following library functions let you move data directly into widgets:

- `sm_putfield` moves the supplied string into the specified widget. If the string is too long, Panther truncates it without warning. If the string is shorter than the destination widget, Panther blank fills it according to the widget's justification. If the data is a null string, Panther clears the field. This refreshes date and time fields that take system values.
- `sm_ww_write` copies text from a string buffer into 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 widget, `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.
- `sm_dtofield` converts a real number value to user-readable format as specified by format. It then moves this value into the specified widget with a call to `sm_amt_format`. If the format string is empty, Panther determines the number of decimal places from the widget's C Type property specification, or from its `numeric_type` property specification. If neither exists, it uses two decimal places.
- `sm_itofield` converts the supplied value to a string and places it in the specified widget.
- `sm_ltofield` converts a long integer passed to user-readable form and places it in the specified widget.
- `sm_amt_format` writes data to a widget, first checking whether the widget has a Data Formatting specification of Numeric. If it does, it formats the data accordingly.

- `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.

Clearing Widget Data

To clear widget data:

- Use the `FERA` logical key to clear a widget's data.
- With the transaction manager, call the `CLEAR` command to clear data throughout, or in a portion of, the transaction tree.
- Use the following library functions to clear data from widgets and arrays:
- `sm_cl_unprot` erases onscreen and offscreen data from all widgets that are unprotected from clearing (`CPROTECT`). Date and time fields that take system values are reinitialized. Widgets with the null edit are reset to their null indicator values.
- `sm_clear_array` clears all data from the array that contains the specified widget. The array is cleared even if it is protected from clearing (`CPROTECT`). `sm_clear_array` and `sm_n_clear_array` also clear arrays synchronized with the array unless they are protected from clearing. Variants `sm_1clear_array` and `sm_n_1clear_array` only clear the specified array.

Inserting and Deleting Occurrences

You can insert rows in using the logical key `INSL` and delete rows using `DELL`.

Two functions let you delete and insert occurrences from arrays:

- `sm_doccur` removes one or more occurrences, starting with the specified occurrence.
- `sm_ioccur` inserts one or more blank occurrences. Before it inserts the occurrences, Panther checks the new total of occurrences is greater than the maximum number of occurrences set for the array.

If other arrays are synchronized with the one specified, `sm_doccur` and `sm_ioccur` perform the same operation on them, provided their Clearing Protect property is set to No. 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.

Both functions ignore the target array's Clearing Protect setting.

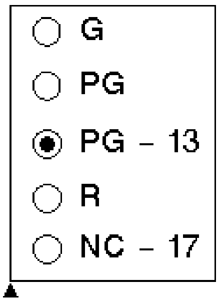
Making Widget Selections

Panther has a set of functions that let you check the current selection or selections within a selection group, and change the selections.

Getting Selections

Two functions, `sm_isselected` and `sm_getfield`, let you determine whether a selection has been made within a selection group and what those selections are.sm_isselected checks whether a selection has been made in a selection group. The selection is referenced by the group name and occurrence number.

If you call `sm_n_getfield` on a radio button group that allows one selection, the buffer that you pass into this function gets the group occurrence number of the selected item. For example, the radio button group rating has the third occurrence, PG-13, selected:

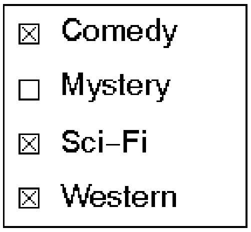


Given this selection, the following call to `sm_n_getfield` puts the string “3” into the string buffer pointed to by `buffer`:

```
ret =
```

```
sm_n_getfield (buffer, "rating");
```

If you call `sm_n_getfield` on a group of widget types that allows multiple selections, for example, a check box group. Panther puts the numbers of the selected occurrences into `buffer`. For example, the `genre` check box group has occurrences 1, 3, and 4 selected:



If you call `sm_n_getfield` on `genre`, `buffer` gets the string 1 3 4.

Panther sees a group's value as an array whose elements contain the offsets of the selected items. Thus, Panther stores the value of `genre` as follows:

```
genre[1] = "1"
genre[2] = "3"
genre[3] = "4
genre[4] = " "
```

`sm_i_getfield` gets the specified selection in the group. For example, this call gets the second-selected item in `genre` and puts its value, 3, into `buffer`:

```
retvar = sm_i_getfield (buffer, "genre", 2);
```

Changing Selections

`sm_select` lets you select an occurrence within a selection widget group. If the group's # of Selections property allows no more than one selection, Panther first deselects the current selection before it selects the specified 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`.

The selected runtime property specifies whether a selection-type widget is selected.

Manipulating Grids

For grid widgets, the `grid_current_occ` runtime property contains the grid widget's current (or selected) occurrence. The following JPL procedure uses this property to delete the selected row.

```
proc delete_selected_row(fld)
vars grid_name occ
{
   grid_name = @widget(fld)->grid
   occ = @widget(grid_name)->grid_current_occ
   call sm_i_doccur(fld, occ, 1)
   return 0
}
```

Making Selections in List Boxes

Inside a list box, you can select multiple occurrences or change the application behavior to only allow a single choice. The `LISTBOX_SELECTION` behavior variable determines the behavior of list boxes in your application with its settings of `SIMPLE_SELECTION` and `EXTENDED_SELECTION` (default). The value for `LISTBOX_SELECTION` must remain constant during the running of the application.

For extended list boxes, the Listbox Type property must be set to Select Any. If it belongs to a selection group, the # of Selections property must also be set to Any. Extended selections pertain to selections within the list box, not the selection group. Therefore, if you have two list boxes in the group, the selection within one list box will have no effect on the other.

One of the field function arguments, `K_EXTEND`, can determine if the widget is an extended list box. Another field function argument, `K_EXTEND_LAST`, can determine if the cursor is in the last item of an extended selection list box.

The `ADDM` logical key toggles in and out of add mode within the list box.

Accessing Tab Controls

A tab control, available for Windows and Motif applications, consists of a tab deck and its associated cards. The tab deck is like a box; any widget within its boundaries is assigned to one of its cards. The `number_of_cards` read-only property gets the number of cards in the deck, including hidden ones.

Each tab card can have widgets which can be grouped as needed. The card property of each widget appearing on a tab card is set to the object ID of that card.

Tab cards are numbered sequentially within the tab deck. The Card Number property (`card_number`) determines the sequence of the cards. You can move to another card by clicking on its index tab or with the `NCARD` and `PCARD` logical keys.

The tab card currently on display in the screen is the topmost card. At runtime, you can set which card is topmost using the tab deck's `topmost_card` property. The following JPL statement set a new topmost card:

```
//  my_deck has three cards named card1, card2, and card3.
//  This call sets the second card as topmost card.

my_deck->topmost_card="card2"
```

Accessing ActiveX Controls

Active X controls, available for Windows and Web applications, are considered separately since the ActiveX control itself is not a Panther' widget, only the ActiveX container is. However, Panther can access the ActiveX control's properties, events, and methods. This allows you to manipulate the ActiveX control at runtime.

The Active X container's `CLSID` property (`clsid`) determines which ActiveX control is located inside the container. If the ActiveX control specified in that `CLSID` property is registered on your system, the control will be displayed in the screen editor and the Properties window will display the control's property names and settings.

To get and set property values at runtime:

- Use the JPL property syntax with `ax_` prepended to the Active X control's property name. The `ax_` prefix insures that there are no conflicts between Panther properties and ActiveX control properties.
- Call the C functions `sm_obj_get_property` and `sm_obj_set_property`.
- For Web applications only, set the properties on the web browser using JavaScript or VBScript.

The author of an ActiveX control specifies what methods can be used to access the control, the arguments needed for those methods, and the events that are applicable to the control. Those methods can then be called in Windows applications using `sm_obj_call` and in Web applications using VBScript or JavaScript. An event handler can then be written for those events using `sm_com_set_handler` in Windows applications and VBScript functions in Web applications.

For more information and an example of an ActiveX control, refer to Chapter 18, “ActiveX Controls,” in *Using the Editors*.

Checking Validation

Panther maintains two runtime properties that can be checked to determine the validation and modification status of a widget or group:

`valided`

Indicates whether or not the widget or group has passed validation:

- `PV_YES`—the widget has passed validation.
- `PV_NO`—the widget has not passed validation.

The valided property is initially set to `PV_NO` when a screen is displayed. It is reset to `PV_NO` each time the content of the widget is changed. It is set to `PV_YES` each time the widget passes its validation tests; for example when `sm_fval` forces validation processing on the widget. You can also explicitly set a widget's valided property to `PV_YES` or `PV_NO`.

During field validation, Panther tests a field's data against a number of formatting and input property settings. Refer to Table 17-1 on page 17-16 for a list of these properties.

In order to allow users to move freely within a GUI application screen, validation is typically suppressed until they explicitly submit the screen data, for example, by pressing a Save push button. Field validation does not typically occur when the user uses a cursor key to move out of the widget, or mouse clicks into another widget. To force validation also to occur on these events, set the application behavior variable `IN_VALID` to `OK_NOVALID`.

For information on validating widgets during screen exit processing, refer to page 17-20, “Screen Exit Processing.”

`mdt`

Indicates whether the data of a widget or group have been modified:

- `PV_YES`—the data is changed.
- `PV_NO`—the data is unchanged.

The mdt property is initially set to `PV_NO` after the screen's entry function is called and is set `PV_YES` when the content of the widget or group changes.

Once set to PV_NO, a widget's mdt property remains unchanged while the screen is displayed; however, you can explicitly reset it to PV_YES.

Panther performs validation on a widget no matter how its valided property is set. If a widget's validation requires significant processing such as a database lookup, you can avoid redundant validation and significant overhead by setting its no_validation property to PV_YES and test the state of its data only when necessary, for example, through the widget's own exit function. The following JPL code tests the mdt property of widget cust_id, to determine whether to force validation processing through sm_fval. Because this widget has validation initially disabled (no_validation = PV_YES), its no_validation property must be reset before the call to sm_fval:

```
/* If the data has changed, force-validate the widget */
if cust_id->mdt == PV_YES
{
  /* reenable validation */
  cust_id->no_validation = PV_NO

  /*  validate widget data */
  if ( sm_n_fval( "cust_id" ) == 0 )
  {
    cust_id->mdt = PV_NO            /* reset mdt flag */
    cust_id->no_validation = PV_YES /* disable validation */
  }
}
```

The following functions let you check and reset the mdt properties for all widgets:

- sm_cl_all_mdts resets the mdt property of all widgets and occurrences to PV_NO.
- sm_tst_all_mdts tests the mdt property of all on- and offscreen occurrences of all widgets on the current screen. If it finds an occurrence with its mdt bit 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 since the screen was displayed or its mdt was last cleared by sm_cl_all_mdts or by resetting the mdt property.

CHAPTER

24 Setting the Screen Sequence

When developing a screen, you need to keep in mind a number of issues relating to the visual appearance of the screen, the type of data being displayed or manipulated, and what your users will do with the data.

These issues can include whether more than one screen can be open at a time. This is a main factor in deciding if you open the screen as a form or as a window.

This chapter discusses the underlying sequence of client screens and how Panther manipulates them.

Forms and Windows

User interaction with a Panther application typically begins with a startup screen, or base form. The base form is often the gateway to other screens, which can be opened in one of two ways:

- As another form. Panther maintains a list of forms, called the form stack. The top form of that stack is the only one that is open.
- As a window that is opened either from the top form or another window. This window can itself open another window, either as a child or a sibling. Panther maintains a stack of all open windows, where the window at the top of the stack

is the active window that is, the window with focus. Panther maintains the window stack only as long as its parent form remains on top of the form stack. The rest of this chapter refers to child windows as stacked windows.

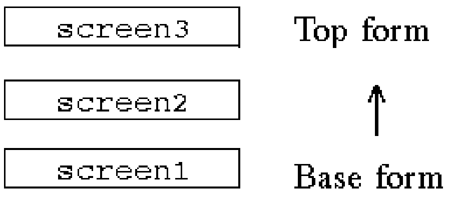
Forms and the Form Stack

Panther maintains a form stack which lists forms previously opened by the application. The application's startup screen is the base form—that is, the first screen to be pushed onto the form stack. Panther pushes onto the stack each screen that is subsequently opened as a form in the application. The top screen is the only form that is open and whose data is accessible.

The form stack retains the names of the screens saved to it and some information about each one's save state—for example, the cursor's last position. However, the stack does not save screen data. Consequently, changes entered earlier on a form might not reappear when the form is reactivated. You can save form data changes through the local data block (LDB). All changes in fields with corresponding LDB entries are written to the LDB when a new form is opened, and can be restored when the earlier form is reactivated. You can also send screen data to a named location in memory, or bundle, for later retrieval; bundles are created and accessed through Panther's send and receive commands and library functions. For more information on LDB processing and send/receive facility, refer to Chapter 25, “Moving Data Between Screens.”

Panther stacks forms in last-in/first-out (`LIFO`) order. All screens in the form stack must be unique. If a form is opened and its name is already in the stack, Panther assumes that you want to return to that form; it pops off the stack all screens above the specified form and discards them.

For example, an application might consist of three screens, `screen1`, `screen2`, and `screen3`, which open each other as forms. This creates a form stack in which screen1 is the application's base form and screen3 is the top form:



screen3 has a menu item that allows users to open screen1 as a form through the control string screen1. On selection of that menu item, Panther finds screen1 already exists in the form stack and returns to that instance. All intervening screens in the form stack—in this case, `screen3` and `screen2`—are destroyed. If the user now exits from screen1, the form stack is empty. If the setup variable `CLOSELAST_OPT` is set to `NO_CLOSELAST`, the application terminates; otherwise, the application continues to run without an open screen.

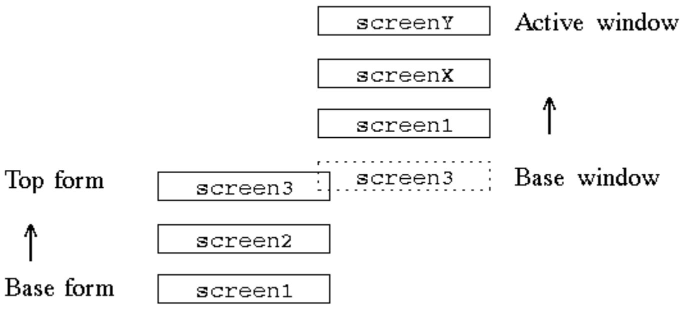
Windows and the Window Stack

The top form always has its own window stack, in which it serves as the base window. Only the top form maintains a window stack. The window stack remains in memory until Panther gets a request to open another form. It then closes all windows and purges that window stack from memory. Finally, it opens the form and creates a new window stack.

Window Stack Organization

Panther stacks windows in last-in/first-out (`LIFO`) order. The top screen in the window stack is the active window and is the only window to have focus. When the application issues a request to close the active window—through `EXIT` or an explicit function call—Panther pops the active window off the window stack. The top window in the stack now becomes the active window with its saved data restored.

For example, given the form stack shown earlier, the top form `screen3` can open screen1 as a window; screen1 can in turn open another window, and so on, yielding a window stack as shown in the following illustration:



In this example, screenY is the top window in the window stack and therefore the active window; only it has focus. If the user closes screenY, for example, through the EXIT key, screenX becomes the active window.

Panther uses the window stack to maintain information about all open windows—which one is active, the order in which they were opened, whether they have a sibling or stacked relationship, and the data of inactive windows. Because the window stack saves inactive window data, Panther can reactivate a window in its previous state, and thus avoids the overhead and processing otherwise incurred by reopening and redisplaying the screen.

The window stack can hold as many windows as system memory allows. You can get the number of windows in the window stack through sm_wcount.

In contrast with the form stack, the window stack can contain multiple instances of the same screen. Be careful to avoid recursive designs that might use large amounts of memory.

Sibling Windows

You can open a screen as a sibling of the current window. Unlike stacked windows, users can bring focus to any window that is a sibling of the active window.

A window cannot directly open any screen as a sibling that is already a sibling. To open multiple instances of the same screen as sibling windows, call sm_setsibling to force sibling status onto the next screen opened as a window. You can also reset an existing window's sibling property (PV_YES to define the window as a sibling, PV_NO to reinstate its stacked window state).

Window Stack Manipulation

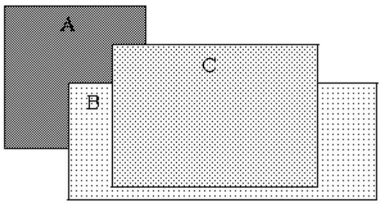
Panther provides these library functions to manipulate the window stack:

- sm_wselect move any window to the top of the window stack; this window becomes the active window. In character-mode, any siblings of the selected window are also brought forward in the display.
- sm_deselect restores a window previously selected by sm_wselect to its former position in the window stack. Panther only saves information about the screen last-selected by sm_wselect call; consequently, you can restore a screen to its previous place in the window stack only if no other windows have subsequently been selected by sm_wselect.

- `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 to open these windows with a call to `sm_setsibling`.

You can also programmatically rotate sibling windows in order to change the active one.

`sm_wrotate` rotates sibling windows according to the supplied step value. For example, the following illustration shows sibling windows A, B, and C, where C is the active window:



The following function call rotates the top sibling window C to the bottom of the sibling stack and leaves screen B on top as the active window:

```
sib_windows = sm_wrotate (1);
```

CHAPTER

25 Moving Data Between Screens

Panther lets you move data between screens by:

- Issuing JPL send and receive commands or their equivalent library functions. These let you explicitly write and read screen data to and from temporary buffers.
- Creating JPL global variables.
- Accessing values on any open application screen.
- Loading and activating screens as local data blocks, or LDBs. LDBs automatically initialize and capture widget data on screen entry and exit, respectively.

Sending and Receiving Data

Panther provides JPL commands and equivalent library functions to transfer data between screens without LDBs. You typically perform send and receive operations as follows:

1. Write data—JPL variables, string constants, and widget values—to a buffer, or bundle.

2. Read data from the bundle into receiving widgets.

advantages over LDBs

Send and receive operations have several advantages over LDB usage:

- A finer level of control over data transfer. You can use any screen- and widget-level hook—entry, exit, and validation—to send and receive data. As developer, you explicitly tell Panther what data to send and when to send it. You also avoid unintentionally overwriting widget data with the LDB, and vice-versa.
- More economical use of memory. This can be especially important in environments with limited memory like MS-DOS.
- Sent data is always delivered to the receiving screen intact. Only the receiving widget length decides whether incoming data is received whole or is truncated.
- Send/receive operations do not require the source and target widgets to share the same name.

The following sections describe send and receive operations in general terms. For detailed information on relevant JPL and library function calls and options, refer to *Programming Guide*.

Bundles

JPL's `send` and `receive` commands and Panther API functions act on bundles, which provide temporary storage for the data you wish to transfer between screens. You can name bundles for explicit access. Panther maintains up to ten bundles by default, including one that can be unnamed. If you send data without specifying a bundle name, Panther writes the data to an unnamed bundle; this data is available to the next receive request that omits specifying a bundle name.

You can set the number of available bundles with the application property, `max_bundles`.

Sending Data

JPL's `send` command and its library function counterparts write screen data to a buffer that is accessible to other screens.

JPL send

JPL's `send` command initializes a bundle and populates it with one or more data items. You can send widget and array values, a specific range of occurrences, variables, and constants.

For example, the following `send` command initializes `bundle1` and sends three data items to it. The third data argument, `credit[1..]`, specifies all occurrences in the array:

```
send bundle "bundle1" data credit_acctno, "1000", credit[1..]
```

Library Function Calls

If you use Panther library functions, you must issue at least three calls in this order:

1. Create a new bundle with a call to `sm_create_bundle`.
2. Create items in the bundle through successive calls to `sm_append_bundle_item`.
3. Populate each bundle item with one or more occurrences of data through `sm_append_bundle_data`. Each call to `sm_append_bundle_data` appends a single occurrence of data to the specified item.

When you are finished sending data to a bundle, you can optionally call `sm_append_bundle_done` to optimize memory allocated for a send bundle.

For example, the following function iterates over all screen-resident widgets and sends their data to the bundle `myBundle`:

```
include <smdefs.h>

int sendScreenDataToBundle(int numFields)
{
   int i, ret;
   if (0 != (ret = sm_create_bundle("myBundle")))
      return ret;
   for (i = 1; i <= numFields; i++)
   {
       sm_append_bundle_item("myBundle");
       sm_append_bundle_data("myBundle",i,sm_fptr(i));
   }
   sm_append_bundle_done("myBundle");
```

```
    return 0;
}
```

Receiving Data

JPL’s receive command and its counterpart sm_get_bundle_data read data from a bundle. The receive command reads bundle data directly into the specified widgets; sm_get_bundle_data reads a single occurrence from the specified bundle item into a buffer and returns with a pointer to that buffer.

JPL receive

JPL’s receive command specifies the bundle to read and reads its data items into the target widgets. For example, the following receive command reads bundle1 and puts its data into three widgets:

```
receive bundle "bundle1" data acctno, credit_amt, credit
```

receive reads data in the same order that it was sent. Because the bundle retains no information about its data sources, the send and receive calls should sequence widgets in the same order to ensure that the receiving widgets get the correct data. Panther does not check whether receive data is valid for the target widgets.

Unless the receive command includes the keep argument, when it returns, Panther destroys the bundle and frees the memory allocated for it. The keep argument keeps the bundle and its data in memory and available for later receive operations.

Library Function Calls

You can use Panther library functions to ascertain a bundle's state and get individual occurrences of data from it. In the next example, sm_get_bundle_item_count and sm_get_bundle_occur_count, respectively, get the number of items in a bundle and the number of occurrences in each item. This example also gets the data from each specified item occurrence through successive calls to sm_get_bundle_data.

```
include <smdefs.h>

/*get the bundle item count and pass it along*/

getNumBundleItems(void)
{
   if !(is_bundle("myBundle"))
      return -1

   getNumOccurs(sm_get_bundle_item_count("myBundle"));
   sm_free_bundle("myBundle");
   return 0;
}

/*get the occurrence count for each bundle item
 *and put occurrence data into screen widgets
 */

void getNumOccurs(int numItems)

{
   int itemCt, oCt, item[numItems];

   for (itemCt = 1; itemCt <= numItems; itemCt++)
   {
      item[itemCt] =
          sm_get_bundle_occur_count("myBundle", itemCt);
      for (oCt = 1; oCt <= item[itemCt]; oCt++)

         /*get data from, each item occurrence, put it into
          *corresponding widget occurrence
          */

         sm_o_putfield
         (itemCt,       /*widget number */
          oCt,          /*occurrence offset*/

         sm_get_bundle_data("myBundle", itemCt, oCt));

}
```

When you finish reading bundle data, destroy the bundle and free its memory by calling `sm_free_bundle`.

Panther also provides these library functions:

- `sm_get_next_bundle_name` gets the name of the bundle created before the one specified. You can use this function to traverse the list of all existing bundles.
- `sm_is_bundle` verifies the existence of a bundle. Use this function to save processing overhead.

Using Global Variables

Use the JPL global command to create variables which can be accessed from anywhere in your application. If desired, you can specify the number of occurrences or an initial value.

The following statement taken from a screen's unnamed procedure creates two global variables: one for the user name and one for an array of divisions.

```
global username, divisions[3] = {"Sales", "HR", "DP"}
```

In Web applications, there are additional types of global variables. For more information, refer to Chapter 7, “JPL Globals in Web Applications,”in the *Web Development Guide*.

Accessing Values on Other Screens

If your application has more than one open screen, you can access a value in another screen using the following syntax:

```
variableName@
screenName
```

For example, the following screen entry procedure writes the value of the user name from a screen named `main.scr` to the current screen:

```
proc screen_entry
call sm_n_putfield(username, username@main.scr)
return
```

Note: In Web applications, this syntax is not available.

Using Local Data Blocks

Panther screens can be used as vehicles for initializing and saving values on other screens. A screen that performs this background role is called a local data block, or LDB. When a screen serves as an LDB, Panther uses its widgets, or LDB entries, to transfer data to and from corresponding widgets on the current screen. By using LDBs, applications can transfer data between screens automatically.

Panther matches LDB entries and screen widgets by name. Only named widgets and LDB entries take part in LDB processing, or write-through. One or more screens can be loaded into memory as LDBs and activated. When Panther enters or exits a screen, it checks whether any LDBs are active. If one or more LDBs are active, Panther performs LDB write-through as follows:

- At screen entry, Panther initializes or overwrites widgets from their matching LDB entries. Screen entry occurs when a screen opens and, optionally, when it is reexposed, depending on the value of `EXPHIDE_OPTION`. In both cases, Panther writes LDB values to the screen after it executes the screen's entry function.

  The LDB always overwrites existing screen data, even if the widget has input protection. One exception applies: at screen open, the LDB respects initial widget data specified through the screen editor—for example, the value set in a push button's Label property. Initial widget data also is written back to the corresponding LDB entry.

- At screen exit, Panther writes data back to the LDB entries. Screen exit occurs when a screen closes and, optionally, when it is overlaid by another screen, depending on the value of `EXPHIDE_OPTION`. Panther writes screen data to the LDB before it executes the screen's exit function.

If data is transferred between arrays, Panther allocates for the target array the number of occurrences required to accommodate the incoming data, up to the array's maximum number of occurrences.

Selection Groups

Panther regards the selections that the user make in a selection group—radio buttons, toggle buttons, check boxes, and list boxes—as the value of that group. You can use LDBs to propagate that value—that is, repeat the selections—from one screen to another. To ensure consistent results, make sure that the screen selection groups and their corresponding LDB entry have the same number of widgets arranged in the same order, and have the same contents.

Restrictions

In general, you should regard LDBs as passive recipients of data. Although an LDB is created and edited as a screen, at runtime, Panther does not perform most of the processing that is otherwise associated with screens. Screen entry and exit functions are not executed; and no validation or formatting is performed on entry data. For example, no updates occur for an LDB entry that is formatted as a date/time widget with `system_update` set to `PV_YES`.

Invalid Targets

Panther does not move values between an LDB and the screen when an error window opens or closes, because these windows do not allow data entry. LDB write-through is invalid for any widget type that is read-only, such as static labels, lines, and boxes.

Data Overflow

LDB entries and their corresponding widgets should have the same data length and number of occurrences. Otherwise, data might be lost for one of these reasons:

- If the length of the target LDB entry or widget is shorter than the source data, Panther truncates the data.
- If the maximum number of occurrences specified for the target LDB entry or widget is less than the number of occurrences allocated for the source, Panther discards the overflow occurrences.

Interaction with Screen Modules

LDB write-through occurs after execution of the screen entry function and the screen module's unnamed procedure. Avoid using either venue to write values directly to widgets if the LDB also writes to those widgets. However, you can circumvent this restriction as follows:

4. Write a procedure or function that populates the widgets with the desired values.
5. Attach this procedure or function to an unused logical key—for example, `APP22=^myproc`.
6. In the screen entry procedure, push this key onto the input queue with the built-in function `jm_keys`. For example:

   ```
   call jm_keys APP22
   ```

After the LDB writes its values to the screen, Panther's screen manager pops all data off the input queue. Given the previous example, when Panther gets `APP22`, it calls `myproc` and executes its contents.

Loading and Activating LDBs

Multiple LDBs can be loaded into memory; of these, one or more can be active at any time. You can activate an LDB only if it is already loaded; only active LDBs are open to read and write operations. If several LDBs are active and have entries with the same name, Panther uses the entry on the most recently loaded LDB. Use

`sm_ldb_get_active` to determine which active LDB is most recently loaded and therefore has precedence.

LDB Handles

Panther assigns a unique integer handle to each loaded LDB. Most runtime functions that access loaded LDBs have variants that let you specify the LDB by its handle or by its name. In this chapter, references to functions use name variants only.

loading multiple instances of an LDB

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.

using displayed screens as LDBs A displayed screen can act as an LDB, but only if it is loaded and activated as such. Note that displayed LDB screens offer numerous opportunities for changing LDB data before it reaches its destination—for example, through user input or widget- and screen-level processing. If you display LDB screens, be careful to safeguard this transitional data.

Default Activation

At application startup, Panther tries to load and activate LDBs as follows:

1. Looks for the configuration variable `SMLDBLIBNAME` and opens all screens in the specified libraries as LDBs.
2. Looks for the configuration variable `SMLDBNAME`. For example:

   ```
   SMLDBNAME = screen1.scr | screen2.scr | screen3.scr
   ```

3. Looks for the library `ldb.lib` and the screens stored in it.

Runtime Loading and Activation

Panther provides several functions for loading and activating, and deactivating and unloading LDBs at runtime:

`sm_ldb_load` loads an LDB into memory and returns its integer handle.

`sm_ldb_state_set` lets you activate a loaded LDB and make it available for LDB write-through. Use this function also to deactivate an LDB; the LDB remains loaded but inaccessible to LDB write-through.

`sm_ldb_unload` removes an LDB from memory, whether active or not.

Read-only LDBs

You can change the state of an LDB from read/write to read-only through `sm_ldb_state_set`. 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. You can use read-only LDBs to maintain constant values for initializing widget data.

push and pop LDBs

Panther has an LDB save stack for push and pop operations. You can remove all loaded LDBs from memory and push them onto the LDB stack with `sm_ldb_push`. Each push operation creates a new entry in the stack, which lists the LDB names and their status—whether active or not. Panther maintains stack entries 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 LDBs to memory, call `sm_ldb_pop`. This function 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.

Getting Information on LDBs

Panther provides several functions that let you get information about loaded and active LDBs and manipulate their values:

- `sm_ldb_get_active` and `sm_ldb_get_next_active` let you iterate over all active LDBs in order of most to least recently loaded. If several LDBs are active, the most recently loaded one has precedence during LDB write-through.
- `sm_ldb_get_inactive` and `sm_ldb_get_next_inactive` let you iterate over all inactive LDBs in order of most to least recently loaded.
- `sm_ldb_state_get` tells whether an LDB is active or whether it is read-only.
- `sm_ldb_is_loaded` tests whether an LDB is loaded.
- `sm_ldb_getfield` gets the current values in an LDB entry.
- `sm_ldb_handle` returns the handle of the specified LDB. Use this with
- `sm_ldb_next_handle` to get the handles of an LDB that is loaded more than once.
- `sm_ldb_name` gets the name of a handle-specified LDB.
- `sm_ldb_putfield` changes the value of an LDB entry.

widget references and LDB entries

Library functions and JPL procedures that reference widgets by name—for example, `sm_n_getfield` and `sm_n_putfield`—seek them first on the screen, then in the LDB. However, on two occasions, Panther reverses the search order: on screen entry or exit, Panther reverses the search order and first looks in the LDB for the requested data. LDB data is written to the screen after screen entry and written back to the LDB before screen exit. Reversing the search order ensures retrieval of the latest data. If the LDB does not contain the requested entry, Panther looks for a corresponding widget on the screen.

You can directly specify LDB entries through `sm_ldb_getfield` and `sm_ldb_putfield` and their respective variants. These functions require you to specify an entry's LDB screen by its name or handle. In JPL, you can reference LDB entries as follows:

```
@ldb( ldb-screen)!ldb-entry
```

CHAPTER

26 Displaying Messages

Panther provides commands and functions that let you display Panther messages in a window or on the status line, or as dialogs. Three-tier applications can send messages between clients and from a server to the client. You can also write a hook function that executes every time one of the error message display functions is called.

This chapter describes the different mechanisms for displaying and handling messages. For information on how to handle errors that occur on a server, refer to Chapter 37, “Processing Application Errors.” For information about event processing and error handling for JetNet and Oracle Tuxedo middleware events, refer to Chapter 6, “JetNet/Oracle Tuxedo Event Processing,” in the *JetNet/Oracle Tuxedo Guide*.

Window Versus Status Line Display

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.

Notes: You can force display of a message to the status line on all GUI and character-mode platforms, regardless of the `MESSAGE_WINDOW` 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.

Acknowledging Messages

Users can dismiss an 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 by embedding the `%Mu` option in the message string (refer to this section on page 45-12).

Disabling Messages

You can control whether the application displays error messages by setting `I_NOMSG` with the `sm_iset` function.

```
proc no_msg
   call sm_iset(I_NOMSG, 1)  //Turn off error messages
return
```

Setting Display Defaults

Several setup variables determine default message presentation and behavior. For more information about these variables, refer to “Message Display” on page 2-20 in the *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. For more information on these, refer to page 45-8, “Setting Message Display and Behavior Options.”

Message Functions

Table 26-1 describes library functions that display errors, messages, and status information. Other functions are related to message storage and retrieval. Message display functions such as `sm_ferr_reset` and `sm_fquiet_err` can either supply a string argument for the message content, or specify a message that is defined in the message file:: For example, this JPL call to `sm_ferr_reset` specifies the string to display in a message window:

```
call sm_ferr_reset (1, "ZIP CODE INVALID FOR THIS STATE.")
```

The next statement supplies a constant (defined in smerror.h) to invoke the application message `Entry is required`.

```
call sm_ferr_reset ( SM_RENRY, @NULL )
```

Application messages are defined in a binary message file and are loaded into memory at initialization. For more information about message files, refer to page 45-2, “Using Message Files.”

Table 26-1 Message related functions

Function	Description
Message display (window or status line):	
sm_femsg	Displays a message and awaits user acknowledgement.
sm_ferr_reset	Identical to sm_femsg when displayed in window. When displayed on status line, puts cursor on at current widget.
sm_fqui_msg	Identical to sm_femsg except that it prepends a tag—for example, ERROR:—to the specified message. Gets the tag from the SM_ERROR entry in the message file.
sm_fquiet_err	Identical to sm_ferr_reset except that it prepends a tag —for example, ERROR:—to the specified message. Gets the tag from the SM_ERROR entry in the message file.
sm_inimsg	Creates a displayable error message on failure of an initialization function For example, if attempts to initialize a message file fail, supply sm_inimsg with the error code returned from the failed function and a description of the function itself. sm_inimsg uses this information to return a string that you can display—for example, by passing it to sm_fqui_msg.
Dialog box display:	
sm_message_box	Displays a message in a dialog box and requests the user to choose a button such as Yes/No, Abort/Retry/Cancel. Pre vents further interaction with the application until the function returns with the user's selection.
Status line display:	
sm_d_msg_line	Can change display attributes of message.
sm_m_flush	Forces display of updates to the status line. Useful if you want to display the status of an operation with sm_d_msg_line without flushing the entire display (e.g., with sm_flush).

Table 26-1 Message related functions

Function	Description
`sm_msg`	Merges the specified message with the current contents of the status line and displays it at the specified column.
`sm_setbkstat`	Saves the contents of a message for display on the status line when there is no other message with a higher priority to display.
`sm_setstatus`	Toggle status line flags. The alternating messages are stored in message file variables `SM_READY` and `SM_WAIT`.
Message file access:	
`sm_msg_get`, `sm_msgfind`	Gets the contents of an application message. Application messages are defined in a binary message file, referenced by the application variable `SMMSGS`.
`sm_msg_read`	Reads into memory a set of application messages from the message file.
`sm_msg_del`	Removes from memory a set of application messages.

Notes: GUI applications should avoid posting message dialog boxes while the mouse button is down. For example, do not call `sm_femsg` from a widget's exit function if the user can mouse click out of that widget into a push button. Doing so can confuse Motif and cause unexpected behavior.

Broadcasting Messages

In JetNet and Oracle Tuxedo applications, two JPL commands enable transmission of unsolicited messages between clients and servers:

- `broadcast` sends a message to all clients that match the specified criteria, or to all clients. Both clients and servers can broadcast messages.

- `notify` sends a message to the client whose service request the server is currently processing. Only servers can use notify to send messages.

Messages are embedded in one of the service message data types that Panther supports, such as `JAMFLEX` or `FML` (Oracle Tuxedo only). For example, the following command broadcasts a message to a client supervisor. It uses source to identify itself as the source of the message:

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

One message handler, installed at application scope, processes all broadcast messages, whether sent by broadcast or notify. The contract for the default message handler `sm_tp_message_print_string` specifies `STRING`-type message data; this data type is valid only in Oracle Tuxedo applications, and limits the broadcast data to a single string. To broadcast complex data, you must write and install a message handler that accepts buffer-type data; to broadcast messages in JetNet applications, this data must be of type `JAMFLEX`.

If your application needs to broadcast messages with variable content and handling requirements, you should write and install a message handler that receives data from a buffer-type message type such as `JAMFLEX`. For example, the following message handler uses the first field of a `JAMFLEX` message to determine the nature of the message and how to handle its contents.

```
// Message handler for all unsolicited messages

proc msg_handler(type, subtype)
vars source, account, date, security, message
vars companyNews, teamNews, stock, stock_quote
vars fileStream, acctMsg

// Identify message sender.

receive MESSAGE ({source})
if (source == "bcast_security")
{
   // receive security violation data
   receive MESSAGE ({account, date, security, message})

   // Alert the supervisor
   msg emsg "%A004Security alert: " ## message ## \
      "%NDate: " ## date ## \
      "%NAccount No. " ## account ## \
      "%NCode: " ## code
}
```

```
else if (source == "bcast_acct_data")
{
   // receive account data
   receive MESSAGE ({account, date, message})
   acctMsg = account##" "##date##"  "##message


   // write message data to log file
   fileStream = sm_fio_open("/u/acct/logfile", "a")
   if fileStream > 0
   {
      call sm_fio_puts(acctMsg)
      call sm_fio_close(fileStream)
   }
}

...

else if (source == "post_comp_news")
{
   // receive posted company news message data
   receive MESSAGE ({ companyNews })
   msg emsg "Latest company news: " ## companyNews
}

...

return 0
```

Status Line Usage

When running in character-mode, Panther reserves one line on the display for error and other status messages. The rightmost part of the status line can display the cursor's current screen position; this can be controlled by calls to `sm_c_vis`.

Message Display

Several types of messages can use the status line; they are described here in order of their priority from highest to lowest.

Error messages

Several functions can be executed to display a message on the status line, wait for acknowledgment from the operator, and then reset the status line to its previous state: `sm_ferr_reset`, `sm_femsg`, `sm_fquiet_err`, and `sm_fqui_msg`. As noted earlier, these functions display messages on the status line only under certain conditions (page 26-1, “Window Versus Status Line Display”). If displayed on the status line, these functions wait for the message to be acknowledged. Messages displayed with these functions have highest precedence.

sm_d_msg_line messages

The library functions `sm_d_msg_line` and `sm_msg` cause the display attributes and message text you pass to remain on the status line until erased by another call to the same function or overridden by a message of higher precedence.

Ready/Wait

The library function `sm_setstatus` provides an alternating pair of background messages. Whenever the keyboard is open for input the status line displays Ready; Wait is displayed when your program is processing and the keyboard is not open. You can change (translate, rephrase, etc.) the display text by editing the `SM_READY` and `SM_WAIT` entries in the Panther message file.

Widget/Menu item status

When the status line has no higher priority text, Panther checks the current widget or selected menu item for text to be displayed on the status line. If the cursor is not in a widget or on a menu item, or if the current widget or item has no status text, Panther uses the string that is set for the screen's `status_line_text` property.

Screen status

When the status line has no higher priority text, Panther checks the current screen's `status_line_text` property for text to display on the status line. If this property is not set, Panther looks for background status text.

Background status

Background status text, the lowest priority of message display, can be set by calling the library function `sm_setbkstat` and passing it the message text and display attributes.

Other Status Line Information

In addition to messages, the status line can hold other information such as cursor position coordinates and debugging information.

The rightmost part of the status line can display the cursor's current screen position as, for example, C 2,18. The display is controlled by calls to `sm_c_vis`.

```
sm_fquiet_err (sm_msg_get (SM_MALLOC));
```

Key constants can be found in the file `smkeys.h` or another of the key header files.

Error Hook Function

Panther calls its installed error function whenever you call one of its error message display routines, such as `sm_fquiet_err` or `sm_ferr_reset`. You can use the error function for special error handling—for example, to write all error messages to a log file. For more information on writing and installing your own hook function, refer to page 44-37, “Error Function.”

Part V Accessing the Database

In Panther, you have different methods of accessing the database. This section gives an overview of database operations as well as instructions on writing SQL statements and using the transaction manager.

CHAPTER

27 Performing Database Operations

Panther provides an interrelated set of tools and software components to help you quickly design applications that perform sophisticated database operations. In order to take advantage of these features, you should have a basic understanding of how database operations are performed in runtime applications, and the steps in your development process which effect these operations.

This chapter describes the different levels of database access that are available in a Panther application.

For information on initializing access to a database engine in an executable, refer to Chapter 7, “Initializing the Database.”

For information on connecting to a database engine, refer to Chapter 8, “Connecting to Databases.”

How Database Operations are Processed

Database operations in Panther applications are processed by the following software components and provide you with different levels of database access:

- Transaction manager—Determines what SQL must be generated and executed, and asks the next level to do the work.
- SQL generator—Constructs SQL statements and asks the next level to execute them.
- Database interface—Passes SQL requests to the database, and returns formatted results to Panther. The database interface is implemented via the `dbms` verb in JPL and the C library function `dm_dbms`.
- Database API—Provided by the database vendor.

Although access via the transaction manager is usually easiest, you can use any combination of levels in your application. For example, you might allow the transaction manager to handle most access itself, but supply specific SQL statements for stored procedure or RPC calls.

Figure 27-1 illustrates how database operations are initiated from the application's event functions by commands that call either the transaction manager or the database interface. The transaction manager relies on additional software components such as the transaction models and the SQL generator to process database transactions.

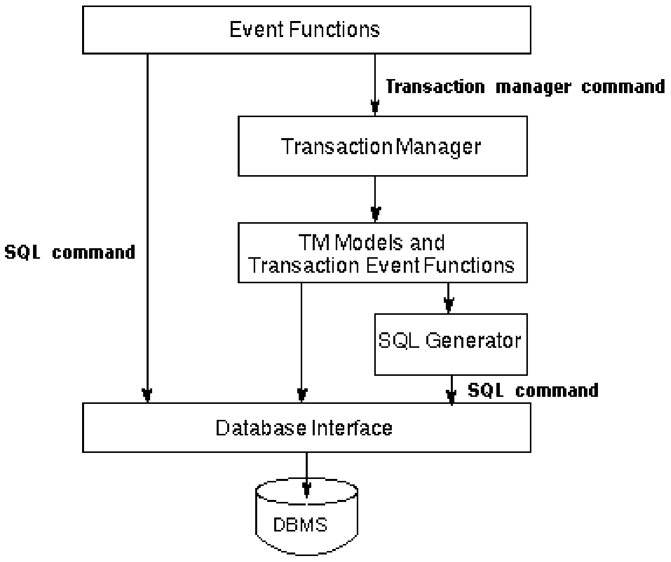


Figure 27-1 The relationship between your application, Panther components, and your database.

Developing Database Operations for your Application

Panther's developments tools such as the screen wizard, screen editor, and repository let you to develop fully functional database applications without writing any code. These tools rely on the transaction manager and the database interface to process your database operations.

However, you are not limited to capabilities provided by the Panther development tools. You can also write your own event functions to directly invoke either the transaction manager or the database interface.

Alternatively or in addition, the database interface has a series of `DBMS` commands which allow you to send SQL statements to the database server and to control how select sets are displayed. You can write your own SQL statements using onscreen values for the database interface to process.

Differences in Application Architecture

In two-tier applications, the client screens can contain:

- SQL statements that are sent to the database.
- Transaction manager commands that generate transaction manager events and subsequent SQL commands.

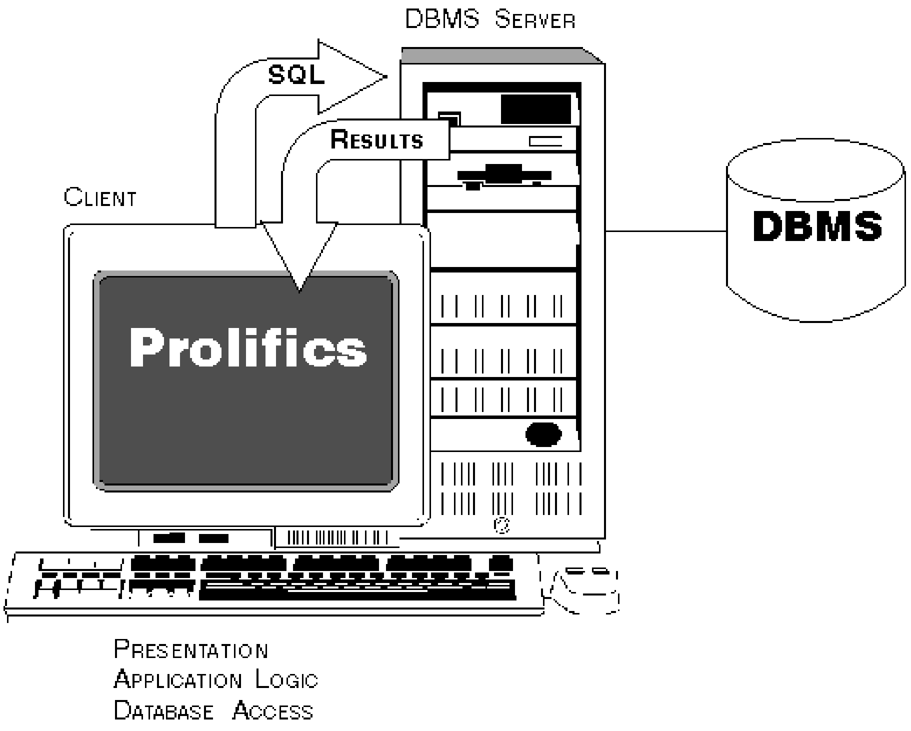


Figure 27-2 In two-tier architecture, each client has direct connection to the database server.

In three-tier applications, client screens can contain service requests or calls to the service component's methods. Both are sent to the application server for processing. For some middlewares, the service requests can be generated by the transaction manager.

On the application server, the transaction manager can generate the events and SQL needed for database operations, or you can write SQL statements as part of the service component.

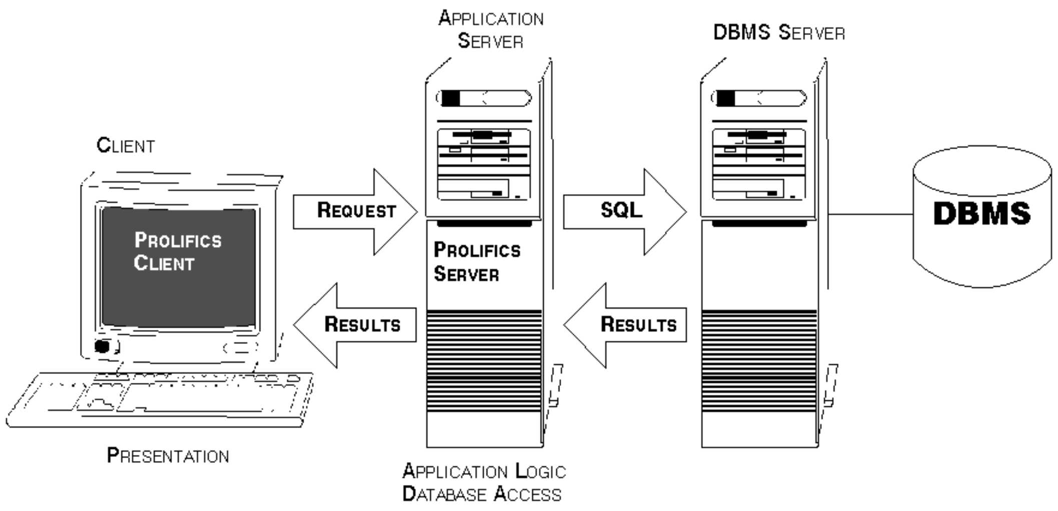


Figure 27-3 Three-tier clients have a single connection to the database by way of the Panther server.

About the Transaction Manager

The transaction manager is a software mechanism that, via property assignments, automatically generates SQL commands for your application's database transactions. In this way, Panther can interact with the database according to user actions. The screen wizard and the screen editor help you develop applications that use the transaction manager to carry out database transactions.

The screen wizard provides transaction manager commands in the event functions associated with the automatically generated screens; however, using just the screen editor, you can copy and manipulate database-derived objects from the repository that contain elements used by the transaction manager. The repository should include most of the information needed for transaction management, either due to the database import or due to custom enhancements made to the repository.

Refer to Chapter 31, “Building a Transaction Manager Screen,” for instructions on building a transaction manager screen.

The transaction manager is controlled by a set of high-level instructions, referred to as transaction manager commands, that are called from the application's event functions (JPL procedures or C functions).

The most common commands are typically invoked from push buttons on your client screens. For example, the `VIEW` command is typically invoked when the user chooses the View button (on a wizard-generated screen), causing the transaction manager to fetch data from a database to display to a user of your application.

After a command is invoked, the transaction manager does suitable traversals of the trees of table views involved in your application, doing the appropriate processing at each table view it reaches. Its default behavior is provided by a distributed common model and database-specific models (collectively referred to as “transaction models”). However, transaction manager processing is ultimately controlled by you, the application developer.

Refer to Chapter 34, “Specifying Transaction Manager Commands,” for an explanation of command syntax.

Refer to Chapter 35, “Generating Transaction Manager Events,” for more information about how the transaction manger generates events for each command.

About the SQL Generator

The SQL generator is called by the transaction models or transaction event functions to generate the appropriate commands to carry out a specific transaction. You can control the composition of the generated commands by setting Database properties for the screen's table views, links, and data entry widgets. The SQL generator uses these property values to form the SQL commands.

Refer to Chapter 33, "Using Automated SQL Generation," for information on how the property settings affect the generated SQL statements.

About the Database Interface

The database interface layer interacts with the database to cause SQL commands to be executed. The database interface (`DBMS`) commands are a set of generalized database-type constructs that allow you to design database-independent applications. These constructs also allow you to specify native SQL commands (using a special syntax), pass them directly through to the database, and, if necessary, pass data back to the client screen.

A special JPL syntax is available for sending onscreen values to the database: colon plus processing. For a description, refer to page 30-1, "Colon Preprocessing."

When the database interface is initialized in your application, it creates two database cursors for application usage: one is used to fetch data, the other to update data. You can use those default cursors for your SQL statements or create new cursors and assign SQL statements to them.

Refer to page 28-3, "Using Database Cursors" for information on declaring new database cursors.

In addition to SQL statements, the database interface commands:

- Map database column names to variables in your Panther application.
- Control database transactions by committing or rolling back a series of SQL statements.
- Write database results to a file.
- Control the appearance of database error messages.

Refer to Chapter 11, "DBMS Statements and Commands," in the *Programming Guide* for a complete description of each `DBMS` command.

Each time a `DBMS` command is executed, it updates a series of status variables. Errors derived from executing commands or SQL statements are displayed through the default error handler. You can also write your own error handler to handle errors or check the status variables. For more information, refer to Chapter 37, "Processing Application Errors."

CHAPTER

28 Writing SQL Statements

You can write the SQL statements needed to access the database for the entire application or for a single screen in the application.

Database Development Process

For database applications where you write the SQL statements, the following steps outline a possible application development process:

Table 28-1 Development process and database operations

Step	Effect on database operations
Import database tables into the repository.	Repository information is used at all stages of screen design and database operation development.
Edit widget properties on repository entries.	Change data formatting and input.
Use the repository entries to create your client screens.	

Table 28-1 Development process and database operations *(Continued)*

Step	Effect on database operations
Edit widget properties on your client screens.	Change data formatting and input.
Write event functions.	Invokes the database interface to perform database operations.
	If needed, create database cursors and assign the SQL statements to those cursors.
	If needed, map Panther variables to database columns.
	Manage database transactions.
Optimize database fetching.	

Database Interface Commands

Two database interface commands can be used to construct your own SQL statements: `DBMS QUERY` and `DBMS RUN`. `DBMS QUERY` is used for `SQL SELECT` statements and for stored procedures that return result set data to the application. For more information on fetching database information, refer to Chapter 29, "Reading Information from the Database."

`DBMS RUN` is used for data modification statements, such as `SQL UPDATE`, that do not return data to the application. For more information on writing information to a database, refer to Chapter 30, "Writing Information to the Database."

Using Database Cursors

A cursor is a SQL object associated with a specific query or operation. Panther stores information on each cursor, including:

- The cursor's name.
- The cursor's connection.
- Any cursor attributes assigned with the following `DBMS` commands: `ALIAS`, `CATQUERY`, `COLUMN_NAMES`, `FORMAT`, `OCCUR`, `START`, `STORE`, and `UNIQUE`.
- Other operation-specific information (for example, the number of rows to fetch, information on target variables or binding parameters, etc.).

Every connection has one or two default cursors which Panther automatically creates. Your application can also declare named cursors on a connection; Panther can use either or both types.

`DBMS` commands are provided for changing the default behavior for a cursor associated with a `SELECT` statement. The commands are `ALIAS`, `CATQUERY`, `COLUMN_NAMES`, `FORMAT`, `OCCUR`, `START`, and `UNIQUE`. For descriptions of these commands, refer to Chapter 11, "DBMS Statements and Commands," in *Programming Guide*.

This section describes the use of cursors, default and named cursors, in an application. For information on how data are passed between an application and a database, refer to Chapter 29, "Reading Information from the Database," and Chapter 30, "Writing Information to the Database."

Using a Default Cursor

Default cursors are convenient for SQL statements that are executed once, and for applications using only one select set at a time. Database commands executed with the JPL commands `DBMS QUERY`, `DBMS RUN`, and `DBMS SQL` use default cursors unless a different cursor is specified.

For most engines, Panther automatically declares two default cursors—one for SQL `SELECT` statements and one for non-`SELECT` statements (such as `UPDATE`). In a few cases, where the engine's standard is a single default cursor, Panther adheres to that standard and declares one default cursor. On such engines, an additional option, `CURSORS`, is supported in the engine's `DECLARE CONNECTION` statement. It permits you to choose between one or two default cursors for the connection. For more information on how cursors are handled for each engine, refer to *Database Drivers*.

A default `SELECT` cursor is associated with a particular connection, namely the connection in effect when a `SELECT` statement is executed. For example:

```
DBMS CONNECTION c2

DBMS WITH CONNECTION c1 QUERY \
   SELECT title_id, name FROM titles \
   WHERE genre_code = 'ADV'

DBMS RUN UPDATE titles SET pricecat = :+pricecat \
   WHERE title_id = :+title_id
```

The first statement sets c2 as the default connection. The second statement uses `WITH CONNECTION` to set c1 as the current connection for the `SELECT` statement. In the `UPDATE` statement, no connection is specified. Therefore, Panther uses the default connection c2.

An application can also close the default cursor if it is not needed. For more information, refer to page 28-9, “Closing a Cursor.”

Using a Named Cursor

Named cursors are convenient for SQL statements that are executed several times. A cursor is declared for a statement; executing the cursor, executes the statement. Named cursors often improve an application's efficiency because the same statement does not need parsing each time it is executed. Named cursors are also necessary for applications using more than one select set at a time.

When a cursor is declared, Panther creates a structure for it and adds its name to a list of open cursors. The cursor is available throughout the application until the application closes the cursor or closes the cursor's connection. Panther frees the structure when the cursor is closed.

You can create one or more named cursors to access and manipulate data. The sequence is as follows:

- Declare one or more named cursors.
- Execute cursor.
- Close cursor.

Declaring a Cursor

Named cursors are created with a declaration statement. The statement names the cursor and associates it with a connection and a SQL statement. If a connection is not named in the declaration, Panther uses the default connection.

```
DBMS [ WITH CONNECTION connectionName ] \
   DECLARE cursorName CURSOR FOR SQL_statement
```

An application can declare a named cursor for any valid SQL statement. For example:

```
DBMS DECLARE c1 CURSOR FOR SELECT * FROM rentals
```

The SQL statement is not executed until the cursor is executed:

```
DBMS WITH CURSOR c1 EXECUTE
```

The cursor can be executed any number of times. The name of the cursor must be a valid Panther identifier. The cursor name is case-sensitive, so `CUR1` and `cur1` are two distinct names.

For more information on writing data to the database and using parameters in a cursor declaration, refer to page 30-11, "Using Parameters in a Cursor Declaration."

Supplying Values Using Colon Expansion

A cursor can use colon-variables in the `DECLARE CURSOR` statement. For example:

```
DBMS DECLARE c1 CURSOR FOR \
   SELECT * FROM rentals WHERE rental_date = :+today
```

The variable `today` is dereferenced when the cursor is declared. It is not dereferenced when the cursor is executed. An application can use colon variables or colon-plus variables anywhere in the statement.

Supplying Values Using Binding

To dereference variables each time the cursor is executed, use bind tags in the `DECLARE CURSOR` statement. For example:

```
DBMS DECLARE c1 CURSOR FOR \
   SELECT * FROM rentals WHERE rental_date = ::rental_date
```

```
DBMS WITH CURSOR c1 EXECUTE USING rental_date = today
```

The bind tag is two colons followed by any valid identifier; the bind tag is not an actual variable. When the cursor is executed, the application must provide a literal value, a valid variable name, or a Panther expression for each bind tag. When this particular example is executed, the application fetches all rentals where `rental_date` is the value of `today`. To execute the select again where `rental_date` is another value, change the contents of `today` and reexecute the cursor:

```
today = @date(today) - 1
DBMS WITH CURSOR c1 EXECUTE USING rental_date = today
```

You can supply a new variable for the bind tag:

```
DBMS WITH CURSOR c1 EXECUTE USING rental_date = yesterday
```

Literals and expressions are valid values for a bind tag. For example:

```
DBMS DECLARE c1 CURSOR FOR \
   SELECT * FROM titles WHERE title LIKE ::title_qbe
```

```
DBMS WITH CURSOR c1 EXECUTE USING title_qbe = "Citizen Kane"
```

or

```
DBMS WITH CURSOR c1 EXECUTE USING \
   title_qbe = title_val ## "%"
```

The first example supplies the literal “Citizen Kane” as the value for the bind tag. The second example uses the concatenation operator (##) to append the contents of the `title_val` variable with the literal percent sign (%) as the value for the bind tag.

It is not required to supply the bind tag names in the `EXECUTE USING` statement. If the tag names are not supplied, Panther associates the first variable with the first tag, the second variable with the second tag, etc. For example:

```
DBMS DECLARE c1 CURSOR FOR \
   SELECT * FROM customers \
   WHERE first_name LIKE ::first_qbe \
   AND last_name LIKE ::last_qbe
```

```
DBMS WITH CURSOR c1 EXECUTE USING f1, f2
```

Panther uses the contents of `f1` as the value for bind tag ::`first_qbe` and uses the contents of `f2` as the value for bind tag `::last_qbe`.

A bind tag is valid for any column value in a `DECLARE CURSOR` statement. A bind tag is not permitted for SQL keywords, table names, or columns names. Therefore, bind tags are valid for column values in any of the following:

- `WHERE` clause of `SELECT`, `UPDATE`, and `DELETE` statements.
- `SET` clause of `UPDATE` statements.
- `VALUES` clause of `INSERT` statements.

For example:

```
DBMS DECLARE c1 CURSOR FOR \
    UPDATE pricecats SET price = ::newprice \
    WHERE pricecat = ::pricecat

DBMS WITH CURSOR c1 EXECUTE USING \
    newprice = price_fld, pricecat = pricecat_fld

DBMS DECLARE c1 CURSOR FOR \
    INSERT INTO pricecats \
    (pricecat, pricecat_dscr, rental_days, price, late_fee) \
    VALUES (::p1, ::p2, ::p3, ::p4, ::p5)

DBMS WITH CURSOR c1 EXECUTE USING \
    p1 = pricecat, p2 = pricecat_dscr, p3 = rental_days, \
    p4 = price, p5 = late_fee
```

Bind tags are also valid for stored procedure parameter values.

Executing a Cursor with Multiple Connections

The command `DBMS EXECUTE` does not permit the `WITH CONNECTION` clause. The cursor remains associated with the connection specified by name or by default in the `DECLARE` statement. For example:

```
DBMS CONNECTION sybcon

DBMS DECLARE cur1 CURSOR FOR SELECT * FROM titles

DBMS CONNECTION oracon

DBMS WITH CURSOR cur1 EXECUTE
DBMS RUN UPDATE ....
```

When cursor cur1 is declared, Panther associates it with the default connection sybcon. Although the default connection is changed to oracon before the cursor is executed, the connection associated with cur1 does not change. When the cursor is executed, Panther performs the SELECT on connection sybcon. The default connection oracon performs the subsequent UPDATE.

Modifying a Cursor

A cursor can be redeclared on the same connection for another SQL statement. For example:

```
DBMS DECLARE abc CURSOR FOR \
   SELECT cust_id, title_id FROM rentals \
   WHERE return_date IS NULL
DBMS WITH CURSOR abc EXECUTE

DBMS DECLARE abc CURSOR FOR \
   SELECT * FROM titles WHERE title_id = ::title_num
DBMS WITH CURSOR abc EXECUTE USING title_num
```

If the cursor is associated with a SQL SELECT statement, you can modify its behavior by using additional DBMS commands These commands include ALIAS, CATQUERY which can be used with FORMAT, COLUMN_NAMES, OCCUR, START, and UNIQUE. Refer to Chapter 11, "DBMS Statements and Commands," in the *Programming Guide* for more information about each command. These settings are not lost when a cursor is redeclared, but only when the cursor is closed. A cursor cannot be redeclared for a different connection.

Using Cursors in the Transaction Manager

Generally, the transaction manager declares and closes cursors as needed. Once the transaction manager creates the select cursor during a TM_GET_SEL_CURSOR event, the variable @tm_sel_cursor contains the name of the select cursor. Using this variable, you can write a transaction event function to declare the cursor and to execute any additional processing. Then, subsequent transaction events attach the SQL statement by redeclaring the cursor.

In the following example, the make_cursor event function declares the cursor and sets a variable to hold select results with the DBMS CATQUERY command. Then, if you choose the VIEW or SELECT command in the transaction manager, this event function is called and is followed by the transaction events that redeclare and execute the cursor with the applicable SQL statement, writing the select results to the `title_all` variable.

In addition, the `RELEASE` command gives up cursors in the transaction manager.

```
proc make_cursor (event)
if event == TM_SEL_BUILD_PERFORM
{
   DBMS DECLARE :@tm_sel_cursor CURSOR
   DBMS WITH CURSOR :@tm_sel_cursor CATQUERY title_all
   return TM_PROCEED
}
return TM_PROCEED
```

Closing a Cursor

Cursors are closed when the application closes the connection. However, if you want to reuse a named cursor, redeclare a cursor on a different connection or free the resources needed by the cursor, you must close the cursor.

To close a cursor and free its data structure, execute:

```
DBMS CLOSE CURSOR cursorName
```

or

```
DBMS WITH CURSOR cursorName CLOSE
```

To close the default cursor, execute:

```
DBMS CLOSE CURSOR
```

The default cursor remains closed unless the application executes a `DBMS QUERY`, `DBMS RUN` or `DBMS SQL` statement without specifying a cursor using the `WITH CURSOR` clause. Panther automatically reopens the default cursor if it is needed.

Database Transaction

A database transaction is a logical unit of work on a database. The unit of work is usually a set of statements that update a database in a consistent way. Either all of the statements in the unit must be completed or none of the statements should be completed at all.

In the VideoBiz sample application, there are a least two transactions:

- A video rental transaction—The first statement in the transaction inserts a row into the rentals table supplying a customer identification code, title code, copy number, rental date, and due date. The second statement of the transaction, updates the customers table for the rental amount and the number of rentals. The third statement updates the tapes table by increasing the number of times the tape (increment of 1) has been rented and changing the status of the rental.
- A new video transaction—Inserts data into four tables: titles, tapes, title_dscr, and roles. The insert into the titles table supplies the name, title code, director information, film length, price category, and film type code. Multiple inserts into the tapes table enters information about each copy of the video. Multiple inserts into the title_dscr table store the film description. Multiple inserts are made into the roles table, each one supplying an actor code and a role for the new video.

Transaction processing is engine dependent and requires an understanding of the engine's behavior. For some errors, the application must explicitly tell the engine to undo the transaction. The application must test for these errors.

Engine-Specific Behavior

Transaction processing is not implemented consistently among SQL databases. Review the documentation on transaction processing supplied by your database vendor.

Database transaction processing, in general, occurs when changes are being made to a database. The transaction begins (with a `DBMS BEGIN` statement), and the changes are not permanently effective until the transaction is committed (with a `DBMS COMMIT` statement). The alternative to committing a transaction is rolling it back (with a `DBMS ROLLBACK` statement), which essentially throws away the changes. Usually, database engines support either explicit transactions, as described here, or auto transactions, which generally start with the first recoverable statement after a logon, `COMMIT`, or `ROLLBACK`.

On engines supporting explicit transactions, each `COMMIT` or `ROLLBACK` must have a matching `BEGIN`. On engines supporting autocommit modes, the application can use any number of `COMMIT` or `ROLLBACK` statements; if there is no recoverable statement, the `COMMIT` or `ROLLBACK` is ignored.

Engines have different ways of handling transactions that are not terminated by an explicit commit or rollback. Some automatically commit or rollback the transaction; others can leave the database in an inconsistent state. Under no circumstances should your application use the engine's default behavior to terminate a transaction.

Use explicit rollbacks and commits to:

- Protect the integrity of the database.
- Make new and updated data available to the rest of the application and other users at the logical end of the transaction.
- Release locks set on tables by the transaction which would otherwise be held until the connection closes, permitting the rest of the application and other users to begin new transactions on the tables.
- Reduce the chances for unrelated operations to interfere with one another.
- Produce applications which are more database-independent.

Finally, although vendors supply commands for transaction processing in their SQL language, use `DBMS COMMIT`, `DBMS ROLLBACK`, and other transaction commands provided with Panther database drivers. Using `DBMS RUN` to specify engine-specific commit and rollback processing is not recommended. Using the `DBMS` versions permits Panther to establish necessary structures and it provides better error handling if a transaction fails.

Error Processing for a Transaction

There are various kinds of errors that can occur during an application. The engine is responsible for recovery from system failures such as power loss. Also, if a single statement fails for some reason in the middle of execution, the engine is responsible for rolling back the effects of that statement. If that statement was executed in a transaction, however, the application must execute an explicit rollback to undo any work done between the start of the transaction and the failed statement.

At the very least, a Panther application must execute a rollback when the engine returns an error to the application. An example of this would be when the engine rejects an insert because the row's primary key is not unique. If the insert were part of a transaction, the application should stop executing the transaction and execute a rollback to undo any work done by previous statements in the transaction.

As an additional precaution, it is recommended that you execute a rollback for any error that occurs during the transaction, including an error detected by Panther before a statement is passed to the engine. An error detected by Panther rather than the engine is usually the result of a development or maintenance error rather than bad user input (for example, a statement's colon-plus or binding variable cannot be found because a widget was renamed). While these errors are rare, the application should provide handling for them.

If the transaction processing is done with the C library functions provided by Panther's database drivers, error codes from Panther are returned to the calling function, either directly or via an installed error handler. If a transaction requires very sophisticated error handling, it might be easier to use these Panther library functions rather than JPL.

One method for transaction processing in JPL uses a generic JPL procedure as a transaction handler. This JPL procedure could perform the following:

- Define and declare a JPL variable, `jpl_retcode`.
- Call a JPL subroutine that contains the actual transaction statements.
- On return from the subroutine, examine the JPL variable, `jpl_retcode`. If it is `0`, the subroutine, and therefore the transaction, executed successfully. If it is not zero, the subroutine was aborted by a Panther or by the error handler. For either type of error, it executes a rollback.

A sample of such a procedure is shown in the JPL code below. The actual transaction statements are executed in the subroutine whose name is passed to this procedure. This transaction handler can be used with the default error handler or with an installed error handler that returns the abort code (`1`) for all errors.

```
proc tran_handle (subroutine)
{
   vars jpl_retcode

# Call the subroutine.
   jpl_retcode = :subroutine

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

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

In this application, there are JPL procedures containing transactions which update the database. The `new_cust` procedure adds a new customer to the database:

```
proc new_cust()
{
   DBMS RUN INSERT INTO customers ....
   DBMS COMMIT
   return 0
}
```

To execute this new customer transaction, the application should execute the following JPL statements:

```
vars newCust = "new_cust()"
call tran_handle (newCust)
```

Once `tran_handle` has set up the variable, it calls the procedure `new_cust`. Whether `new_cust` is successful or unsuccessful, control is always returned to `tran_handle`.

Refer to the *Database Drivers* for a list and description of the supported transaction commands for each engine.

CHAPTER

29 Reading Information from the Database

The database interface provides access to a database engine through one of Panther's database drivers. You can enter SQL statements using the SQL syntax supported by your database engine. With the database interface, you also have access to a series of commands to help you return the information to Panther variables. These commands are included in each of Panther's database drivers and can be used in either JPL procedures or C functions.

A Panther application receives two types of information from a database:

- Data requested by a `SELECT` statement.
- Error and status codes.

This chapter discusses how this information flows from one or more databases to variables in a Panther application, in particular the destination and format of data returned by `SQL SELECT` statements. For information about error and status codes, refer to Chapter 37, “Processing Application Errors.”

The `SQL SELECT` statements would be part of a client screen in a two-tier application and a service component in a three-tier application. In a three-tier application, client screens obtain database information by sending service requests to the application server. For information about writing service requests in a JetNet or Oracle Tuxedo application, refer to Chapter 5, “Defining Services in JetNet and Oracle Tuxedo Applications,” in the *JetNet/Oracle Tuxedo Guide*.

An application can also receive data as the result of executing a stored procedure. Since all engines do not support stored procedures, and the syntax of commands varies among those that do, refer to the *Database Drivers* for more information.

The information on how data is mapped to Panther variables also applies to processing in the transaction manager even though most of the examples in this chapter use the `DBMS QUERY` command to construct the `SQL SELECT` statements.

Fetching Data Using SELECT Statements

When a `SELECT` statement is passed to an engine, Panther performs several steps before transferring data to Panther variables.

1. Panther counts the number of columns in the query and records information on each column's name, length, and data type, noting whether it is a character, date or numeric data type.
2. For each column, it searches for a Panther variable destination. If a destination exists, Panther records the length of the variable. If no Panther destination exists for a column, or if the destination is an LDB variable with initial content, Panther does no fetches for the column. Refer to the following section for more information on Panther destinations.
3. It determines the number of rows to fetch. This number usually equals the number of occurrences in the smallest Panther destination variable, or `0` if there are no target variables. Refer to “Fetching Multiple Rows” on page 29-8 for more information.
4. Finally, Panther formats data before writing it to the destination variables if the database column has a date data type, or if the destination variable has a null, currency, or precision property specification. Refer to “Format of Select Results” on page 29-15 for more information.

The sequence above describes a `SQL SELECT` that writes database column values to occurrences of a widget, JPL variable, or LDB variable. You can also direct the results of a `SELECT` to a text file or concatenate all the values in a row to a single Panther variable. Refer to page 29-19 for more information.

Targets for a SELECT Statement

For an application to retrieve data from a database, there must be an unambiguous mapping between a selected database column and its Panther destination. There are two ways of associating Panther target variables with database columns.

- Give a Panther target variable the same name as a database column. This is called automatic mapping.
- Explicitly declare a Panther variable as the target of a database column. This is called aliasing.

Automatic Mapping

By default when executing a `SELECT` statement, Panther will search for variables with the same names as the specified columns. These Panther variables can be widgets, JPL variables, or LDB variables. For the statement,

```
DBMS QUERY SELECT title_id, name, pricecat FROM titles
```

to return values to Panther variables, the table `titles` must have at least three columns: `title_id`, `name`, and `pricecat`. If any of these columns does not exist in the table `titles`, the engine returns an error.

The application can have a Panther destination variable for none, some, or every named column in the `SQL SELECT` statement. To return the values of all three columns to the application, there must be a Panther variable for each column. The variables can be named `title_id`, `name` and `pricecat`. If one of these variables does not exist, Panther ignores the values belonging to that particular column.

Panther also permits the use of the * in the `SELECT` statement,

```
DBMS QUERY SELECT * FROM titles
```

Using automatic mapping, Panther looks for a variable for each column in the table `titles`. Columns without matching variables are simply ignored. This is not treated as an error.

using qualified column names

You can use one or more qualified column names in `SELECT` statements. For example,

```
DBMS QUERY SELECT titles.title_id, titles.name,
   titles.pricecat FROM titles
```

The Panther targets, however, must be given unqualified names: `title_id`, `name`, and `pricecat`.

matching the engine's case flag

When using automatic mapping, the case of the Panther variable names should correspond to the case flag used in the engine initialization. If the engine's case flag is `DM_FORCE_TO_LOWER_CASE`, the Panther variables for a `SELECT` statement should have lower case names. If the case flag is `DM_FORCE_TO_UPPER_CASE`, the Panther variables should have upper case names. If the case flag is `DM_PRESERVE_CASE`, the Panther variables should match the exact case of the database columns. For information on a particular engine's case flag, refer to the *Database Drivers*.

Aliasing

Aliasing is used when automatic mapping is inconvenient or impossible to use. In particular, aliasing is necessary when selecting any of the following:

- A column whose name is not a legal Panther variable name.
- A column whose name conflicts with other Panther variable names in the application.
- A computed column or the result of an aggregate function (e.g., `COUNT`, `SUM`, `AVG`, `MAX`, `MIN`).

Aliasing is not limited to these conditions. Any or all columns can be aliased if desired. For example, you can alias a column if its name is not descriptive or if you wish to name target variables for a particular table and column.

Panther provides the command `DBMS ALIAS` to specify aliases. On some engines, you can also use the engine's `SELECT` syntax to specify aliases.

Using DBMS ALIAS

`DBMS ALIAS` is associated with a `SELECT` cursor, either a named cursor or the default `SELECT` cursor. If a cursor is not named, the aliases affect all `SELECT` statements executed with the default cursor. You can assign aliases by name or by position. The following syntax aliases a column name to a Panther variable:

```
DBMS [WITH CURSOR cursor] ALIAS column1 pantherVar1 \
   [, column2 pantherVar2 ... ]
```

The following syntax aliases a column position to a Panther variable:

```
DBMS [WITH CURSOR cursor] ALIAS pantherVar1 [, pantherVar2 ... ]
```

Only one `DBMS ALIAS` statement can apply at any one time to any named or default cursor. In that statement, either named or positional aliasing can be used, but both forms can not be used in a single `DBMS ALIAS` statement.

turning off aliasing

To turn off aliasing, execute `DBMS ALIAS` without any arguments. Again, if a cursor name is given, aliasing is turned off on the named cursor. If no cursor name is given, aliasing is turned off on the default cursor.

The case of the column names in the `DBMS ALIAS` statement should correspond to the case flag used in the engine initialization. If the engine's case flag is `DM_FORCE_TO_LOWER_CASE`, the column names should be in lower case. If the case flag is `DM_FORCE_TO_UPPER_CASE`, the column names should be upper case. If the case flag is `DM_PRESERVE_CASE`, the column names should use the exact case of the database columns. The case of *`pantherVar`* should always match the exact case of the Panther variable name. For information on a particular engine's case flag, refer to the *Database Drivers*.

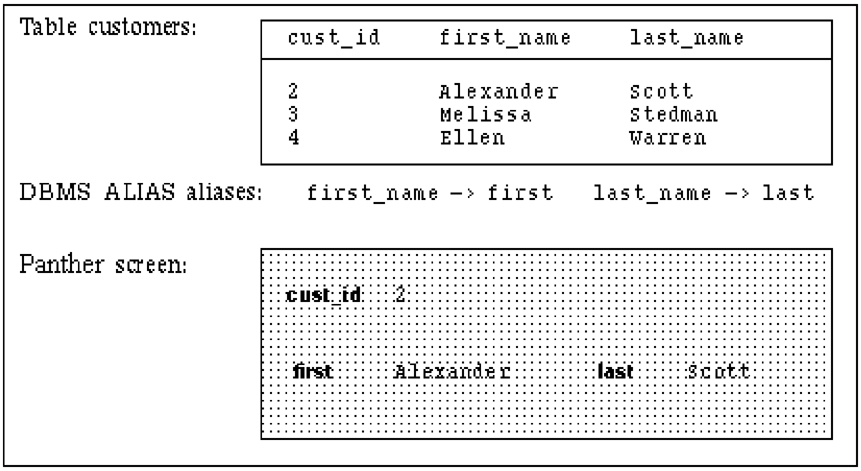
If an application aliases a column to a Panther variable that does not exist, Panther ignores the column's values. This is not treated as an error.

Aliasing by Column Names

First, consider an example that aliases column names to Panther variables. For example:

```
DBMS ALIAS first_name first, last_name last
DBMS QUERY SELECT cust_id, first_name, last_name FROM customers
```

Panther writes the values from the column `first_name` to the variable `first` and it writes the values of column `last_name` to the variable `last`. Since no alias was given for `cust_id`, it maps it to a variable of the same name. This is illustrated in Figure 29-1.



cust_id	first_name	last_name
2	Alexander	Scott
3	Melissa	Stedman
4	Ellen	Warren

Figure 29-1 The mapping of a SELECT statement when aliases are used.

Aliases can also be given after declaring a named cursor. For example:

```
DBMS DECLARE cust_cursor CURSOR FOR SELECT \
   cust#, member_date, member_status FROM customers
DBMS WITH CURSOR cust_cursor ALIAS "cust#" cust_num
DBMS WITH CURSOR cust_cursor EXECUTE
```

Since `cust#` is not a legal Panther variable name, the application must declare an alias for the column if it is to receive the column's value. Before executing the cursor, the application aliases column `cust#` to variable `cust_num`. The cursor keeps this alias until the application turns it off with `DBMS ALIAS` or closes the cursor with `DBMS CLOSE CURSOR`. If a column name is not a valid Panther identifier, enclose it in quote characters; this ensures that Panther parses it correctly.

Aliasing by Column Positions

Consider an example that uses positional aliases. For example:

```
DBMS ALIAS min_rent, max_rent, avg_rent
DBMS QUERY SELECT MIN(num_rentals), MAX(num_rentals),
AVG(num_rentals) FROM customers
```

Panther writes the aggregate function values to the alias variables. The value of `MIN(num_rentals)` is written to the variable `min_rent`, `MAX(num_rentals)` is written to the variable `max_rent`, and `AVG(num_rentals)` is written to the variable `avg_rent`. There is no automatic mapping available for values resulting from calculations or aggregate functions. If the application had not declared aliases, the values would not be written to Panther variables.

Of course, the application should turn off the positional aliases when it is finished. If it does not turn them off before executing the next `SELECT` statement on that cursor, Panther will attempt to write the values of the first three columns to the three positional alias variables. If those variables are no longer available, Panther will ignore the first three columns in the select set.

Aliasing with the Engine's SELECT Syntax

Many engines support aliasing in their `SELECT` statement syntax. In interactive mode, this permits the user to specify for a view a column heading that is different than the database column name. Typically, the syntax is

```
SELECT column1  heading1, column2  heading2...FROM table
```

In interactive mode, the values of `column1` are placed under the heading `heading1`, and the values of `column2` are places under the heading `heading2`. In this syntax, a space separates a column from its alias, and a comma separates the column-alias set from the next column or column-alias set. Some engines might support another syntax. Refer to your database engine documentation for details.

If an engine supports aliasing in a `SELECT` statement, Panther will also support it. You can follow the syntax of the engine, replacing heading with the name of the appropriate Panther variable.

For example, if the syntax shown above is supported by the engine, than the following could be used in a Panther application,

```
DBMS QUERY SELECT title_id id, name, pricecat price
 FROM titles
```

When this statement is executed, the DBMS tells Panther that the columns `id`, `name`, and `price` were selected. Panther looks for variables with those names. If there is a variable `title_id` available, this `SELECT` statement will not write to it because the engine has aliased it to `id`.

Although this form is supported, using `DBMS ALIAS` is recommended, especially for applications accessing more than one engine. Panther provides identical support for `DBMS ALIAS` on all engines.

Fetching Multiple Rows

Since a select set often contains more than one row, you must specify how many rows to fetch at one time. In addition, the application architecture determines whether you can fetch additional rows from the database. In two-tier applications with a direct connection to the database, `DBMS CONTINUE` commands can be used to fetch subsequent rows. In three-tier applications, all rows must be fetched at once.

Determining the Number of Occurrences

Panther uses the following guidelines in determining the number of rows to fetch:

- If an occurrence number was specified with a target variable name, only one row is fetched.
- If a target is a multitext widget with the `word_wrap` property set to `PV_YES`, only one row is fetched.
- If using browse mode, only one row is fetched. (Refer to Database Drivers to see if the engine supports browse mode.)
- The number of rows that the database interface will return to arrays with an unlimited number of occurrences is controlled by the `max_fetches` property. The default value is 1000. You can increase or decrease this value. Changing the property affects any subsequent `SELECT` or `CONTINUE` statements.

- Otherwise, Panther examines the number of occurrences in each of the targeted variables. Usually, all the target variables have the same number of occurrences. If this is true, Panther fetches a row for each occurrence. If the targets do not have the same number of occurrences, Panther finds the target variable with the least number of occurrences and fetches that number of rows.

  Therefore, if the targets for the `SELECT` statement contain both a single line text widget and an array, only one occurrence is fetched. Similarly, if the target variables are multitext widgets and one of those widgets has the `word_wrap` property set to `PV_YES`, only one occurrence is fetched for the entire set.

Be careful of LDB variables that are unintentional targets of a `SELECT` especially when using the wild card * in a `SELECT` or when executing a `SELECT` in a screen entry function.

For example, consider an application using the wild card:

```
DBMS QUERY SELECT * FROM table
```

The application has onscreen widgets for some of the columns in the table. The LDB, however, contains an entry with the name of one of these unrepresented columns. If the onscreen fields have 20 occurrences and the LDB entry has 5 occurrences, only five rows will be fetched.

Also, consider an application that executes a `SELECT` in a screen entry function. By default, Panther first searches the LDB and then the screen for Panther variables when executing screen entry functions. Therefore, if a variable is represented both as an onscreen field and as an LDB variable, a screen entry function will write to the LDB variable before the LDB merge writes to the onscreen field. If the LDB variable and the field do not have the same number of occurrences, data is lost or appears lost when the LDB merge updates the screen fields.

Scrolling Through a SELECT Set

Most applications must be capable of handling a fluctuating number of data rows. Based on the type of data selected and the hardware in use, you can use either or both types of scrolling—scrolling arrays or non-scrolling arrays.

If scrolling arrays are used as the destination variables of a `SELECT` statement, the entire select set is fetched in a single step. To view the rows, press the page up and page down keys (logical keys `SPGU` and `SPGD`).

Otherwise, the application uses single-element fields or non-scrolling arrays as the destination variables of a `SELECT` statement. The select set is fetched incrementally. To permit the user to scroll backward and forward in the set, the application must set up a method to execute the Panther scrolling commands.

The two methods are described in detail below.

Using Scrolling Arrays

Scrolling arrays are useful for small to mid-sized sets. Set the scrolling property (under Geometry) to `PV_YES`. Set the # of Occurrences (`num_occurrences`) property or leave it blank, in which case the number of occurrences is determined by the `max_fetches` property. By default, this property is set to 1000. Because the application must keep the entire select set in memory, you might want to lower the maximum number of occurrences depending on the platform or for a `SELECT` involving many columns.

With this approach, you create large scrolling arrays with more occurrences than the number of rows you expect to be in the select set. When the `SELECT` is executed at runtime, there is no penalty for unused occurrences; Panther allocates only whatever memory is needed to hold the returned rows. Therefore, a Panther screen might contain variables each with 10 elements and 1000 occurrences. If a select set contained only 75 rows, Panther would allocate memory for 75 occurrences in each of the variables; it would not allocate memory for the 925 unused occurrences.

There are several ways of verifying that the arrays actually contained enough occurrences to hold the entire select set. Most often the application examines the value of the global variable `@dmretcode`. Panther writes a no-more-rows status code to this variable when the engine signals that it has returned all requested rows. The value of this variable can be examined after a `SELECT` statement.

Refer to Table 37-1 for more information on `@dmretcode` and related variables. An example procedure is shown below:

```
proc select_all
DBMS QUERY SELECT cust_id, first_name, last_name, member_status
                                             FROM customers
if @dmretcode == DM_NO_MORE_ROWS
   msg esmg "All rows returned."
else
   msg emsg "Application could not display all customers."
return
```

This approach is very easy to use. Since all the rows are fetched at once, the application makes only one request of the database server and it is free to use the default SELECT cursor to make new selects.

It is not the best method for large SELECT sets. If the application is too slow displaying the data or is sluggish after the rows have been fetched, you should consider using non-scrolling arrays or some other alternative scroll driver.

Using Non-scrolling Arrays

Non-scrolling arrays are useful for mid-sized to large select sets. Panther does not impose any limit on the number of rows that can be displayed with this method.

For widgets to be non-scrolling arrays, the array_size property is set to > 1 and scrolling is set to PV_NO. At least two JPL procedures are needed to view the select set. The first procedure executes the SELECT statement and fetches the first screenful of rows. The second procedure executes a DBMS CONTINUE to fetch the next screenful of rows from the select set. The second procedure might be executed many times before the user sees all the rows.

Notes: In multi-user environments, you should know how the engine ensures read consistency—the guarantee that data seen by a SELECT does not change during statement execution. The engine might be using rollback segments or shared locks to provide read consistency. Since a shared lock prevents other users from updating locked rows, applications on these engines should release the lock as soon as possible.

For example, the current screen has widgets named for the columns in the table titles. Each widget has array_size set to 5. The application uses procedures like the following to select data from a table and view additional rows:

```
proc select_video
DBMS QUERY SELECT * FROM titles
return

proc continue_select
DBMS CONTINUE
return
```

as well as control strings like the following to execute the procedures:

```
PF1=^select_video
PF2=^continue_select
```

Assume that table titles contains 12 rows. When you press the PF1 key, the application executes the JPL procedure `select_video` and writes rows 1 through 5 to the screen. If you press PF2, the application executes the procedure `continue_select` which clears the arrays and writes rows 6 through 10 to the screen. If you press PF2 again, the application executes `continue_select` again which clears the arrays and writes rows 11 and 12 to the screen. If you press PF2 a third time, the application does nothing because there are no more rows in the select set.

Non-scrolling arrays use less memory than scrolling arrays. With non-scrolling arrays, the application needs only enough memory for the rows displayed on screen. The other rows are buffered either in a binary disk file or by the database server. With large select sets, this approach often improves the application's performance and response time.

This approach requires a little more work. The application needs procedures to handle the scrolling and possibly the remapping of cursor control keys. Also, the method restricts the `SELECT` cursor. If the application needs to perform other `SELECT` statements while scrolling through this set, the application must declare named cursors to execute additional SQL statements.

Scrolling Commands

In addition to `DBMS CONTINUE`, an application can simulate scrolling through a `SELECT` set by using the following commands:

`DBMS CONTINUE_BOTTOM`	Scrolls to the last screenful of rows
`DBMS CONTINUE_DOWN`	Does the same as `DBMS CONTINUE`
`DBMS CONTINUE_TOP`	Scrolls to the first screenful of rows
`DBMS CONTINUE_UP`	Scrolls up a screenful of rows

Some engines have native support for these commands. For example, the engine might buffer the rows in memory on the server. However, Panther also provides its own support for these commands. Use `DBMS STORE FILE` to set up a continuation file for a named or default `SELECT` cursor. When it is used, Panther buffers `SELECT` rows in a temporary binary file. The syntax of the command is:

```
DBMS [WITH CURSOR cursor] STORE FILE [filename]
```

The command is supported on all engines. To select and view data, an application uses procedures like the following:

```
proc select_video
DBMS STORE FILE vidlist
DBMS QUERY SELECT * FROM titles
return

proc scroll_down
DBMS CONTINUE
return

proc scroll_up
DBMS CONTINUE_UP
return

proc scroll_top
DBMS CONTINUE_TOP
return

proc scroll_end
DBMS CONTINUE_BOTTOM
return
```

Then, you attach these procedures to push buttons or function keys. The following example attaches the procedures to function keys:

```
PF1=^select_video
PF2=^scroll_down
PF3=^scroll_up
PF4=^scroll_top
PF5=^scroll_end
```

Using the same number of rows and occurrences as earlier, when you press the PF1 key, the application executes the JPL procedure `select_video` and writes rows 1 through 5 to the screen. If you press PF2, the application executes the procedure `scroll_down` which clears the arrays and writes rows 6 through 10 to the screen. If you press PF3, the application executes `scroll_up` which clears the arrays and writes rows 1 through 5 to the screen. If you press PF5 the application executes `scroll_end` which clears the arrays and writes the last 5 rows in the `SELECT` set, rows 8 through 12, to the screen.

Remapping Logical Keys for Scrolling

Instead of using function keys or push buttons to call the JPL procedures which execute the Panther scrolling commands, you might prefer the standard page up and page down keys to the PF keys. The values of the logical keys `SPGU` and `SPGD` can be reassigned with the Panther library function `sm_keyoption`. Therefore, the application might use an entry and exit function to change how `SPGU` and `SPGD` work on a screen or in a field. The entry function calls `sm_keyoption` so that SPGD acts like the function key that calls the scroll up procedure, and calls `sm_keyoption` so that `SPGU` acts like the function key that calls the scroll down procedure. The exit function calls `sm_keyoption` to restore the default behavior.

An example of this behavior in a widget entry and exit function is shown below. The widget's `entry_function` and `exit_function` properties are set to `entry_exit` which calls `sm_keyoption`. The function keys `APP1` and `APP2` are set to call the JPL procedures `scroll_up` and `scroll_down` described above. When you click on the widget, the standard page up and page down keys can be used to scroll through the data.

```
// APP1=^scroll_up
// APP2=^scroll_down

proc entry_exit(f_no f_data, f_occ, f_flag)
    if (f_flag & K_ENTRY)
    {
        call sm_keyoption (SPGD, KEY_XLATE, APP1)
        call sm_keyoption (SPGU, KEY_XLATE, APP2)
    }
    else if (f_flag & K_EXIT)
    {
        call sm_keyoption (SPGU, KEY_XLATE, SPGU)
        call sm_keyoption (SPGD, KEY_XLATE, SPGD)
    }
return
```

Controlling the Number of Rows Fetched

If you use widget or LDB arrays as the destinations of a `SELECT`, you can specify the maximum number of rows to fetch and the first occurrence to write to in the array destination. The command is

`DBMS [ WITH CURSOR` *`cursor-name`* `] OCCUR` *`int`* `[ MAX` *`int`* `]`

`DBMS [ WITH CURSOR` *`cursor-name`* `] OCCUR CURRENT [ MAX` *`int`* `]`

Refer to page 11-38 in the *Programming Guide* for more information on this command.

Choosing a Starting Row in the SELECT Set

You can also change the number of rows fetched by using the command

```
DBMS [ WITH CURSOR cursor-name ] START int
```

The command tells Panther to read and discard `int` - 1 rows before writing the rest of the select set to Panther variables.

Refer to page 11-50 in the *Programming Guide* for more information on this command.

Format of Select Results

Before writing a database column value to a Panther variable occurrence, Panther determines the data type of the database column.

In all cases, if the value equals the engine's null (for example, `NULL`), Panther clears the variable. If the variable has the `null_field` property set to `PV_YES`, Panther automatically converts the null string to the one assigned by the widget's property specification.

If any value is longer than the variable, the data is truncated.

Character Column

If a column has a character data type, the value is simply written to the target variable. If the variable has the `word_wrap` property set to `PV_YES` or the justification property set to `PV_RIGHT`, the property is applied.

Date-time Column

If a column has a date data type, Panther formats the value before writing it to a Panther variable. If the variable has a date-time specification, Panther uses it. If the variable does not, Panther uses the format assigned to the message file entry `SM_0DEF_DTIME`. By default, the entry is

```
SM_0DEF_DTIME = %m/%d/%2y %h:%0M
```

For example, April 1, 2015 10:05:03 would be formatted as 4/1/15 10:05. When the message file default is used, Panther assumes a 12-hour clock.

For information on date and time formats, refer to page 10-17 in the *Using the Editors*.

Numeric Column

If a column has an integral type, Panther converts the value to a long. Panther then converts the value to ASCII and writes it to the variable, truncating any data longer than the destination variable. If `data_formatting` is set to `PV_NUMERIC` and `c_type` is set to `PV_DEFAULT`, the numeric format is applied to the data.

If a column has a real type, Panther converts the value to a double. Before writing the value to a Panther variable, Panther examines the widget's Data Formatting and C Type properties to help determine the precision.

- Numeric format (`data_formatting = PV_NUMERIC`) and default data type (`c_type = PV_DEFAULT`)

  If the value is less precise than that specified in the Min Decimal (`min_decimals`) property (defines the minimum number of decimal places), the value is padded to the minimum number of decimal places. If the value is more precise, it is rounded or adjusted to the numeric type's maximum number of decimal places (`max_decimals` property). The Rounding (rounding) property specification (round up, round down, or adjust) of the numeric format is applied.

- None numeric data (`data_formatting = PV_NONE`) and the data type is either float, double, integer, long integer, short integer (`c_type = PV_FLOAT | PV_DOUBLE | PV_INT | PV_LONG_INT | PV_SHORT_INT`)

  If the C type is one of the integer types, the value is adjusted by standard rounding to 0 places. If the C type is float or double, the value is padded or adjusted to the type's precision.

- Data Formatting, C Type and Precision properties conflict
  - If the value is less precise than the numeric format's number of decimal places (`min_decimals`), the value is padded to the number of decimal places specified.
  - If the value is more precise than the numeric format's number of decimal places (`max_decimals`), Panther compares the numeric format of places and the C type's precision, and uses the less precise of the two. If the C type precision is less precise, the numeric property specifications still controls how the value is displayed and therefore, the value is padded if necessary.
  - If it uses the numeric format's maximum number of places (`max_decimals`), then it also uses the Rounding property specification (round up, round down, or adjust) as well as any fill characters (`fill_character`).
  - If it uses the C type precision specification, it adjusts by standard rounding to the precision.
- None numeric data (`data_formatting = PV_NONE`) and a default data type (`c_type = PV_DEFAULT`).

  The precision is taken from the data type being returned.

Refer to page 10-20 in the *Using the Editors* for more information on currency formats.

Binary Columns

If a column has an binary data type, Panther sets the column type to be `DT_BINARY`. Before writing data to the widget, Panther checks the `c_type` property. If the setting is `PV_HEX_DEC`, then Panther converts the binary data to a hexadecimal string. Otherwise, Panther passes the binary data as is.

Generally, binary data is fetched either into variables declared with `DBMS BINARY` or into widgets with a `c_type` property of `PV_HEX_DEC`. Otherwise, incorrect binding might result.

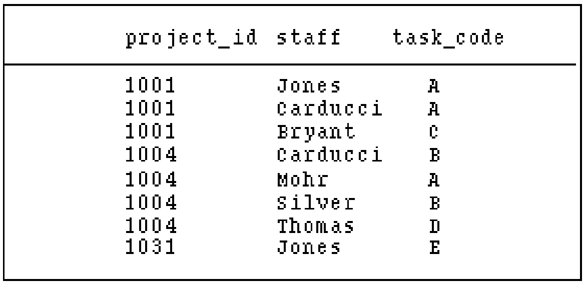
Fetching Unique Column Values

By default, when a column is selected, Panther returns all values. There is also a command for displaying only a column's unique values,

```
DBMS [ WITH CURSOR cursor-name ] UNIQUE column [, column ... ]
```

Panther replaces a repeating value with an empty string.

This command is useful if an application is selecting values from a table which uses two or more columns as the primary key. For example, if the table projects has the columns `project_id`, `staff`, `task_code` and the columns `project_id` and staff constitute the primary key, an application could suppress the repeating values in one of the columns of the primary key to improve readability on the screen. Figure 29-2 illustrates the data in the project table.



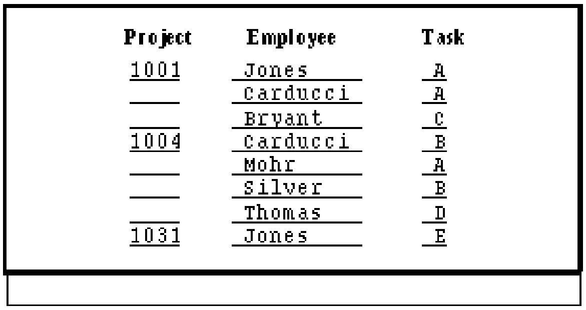
project_id	staff	task_code
1001	Jones	A
1001	Carducci	A
1001	Bryant	C
1004	Carducci	B
1004	Mohr	A
1004	Silver	B
1004	Thomas	D
1031	Jones	E

Figure 29-2 The primary key of the projects table is (project_id, staff).

The following commands select the data and format it to suppress repeating values:

```
DBMS DECLARE proj_cur CURSOR FOR \
   SELECT * FROM projects ORDER BY project_id
DBMS WITH CURSOR proj_cur UNIQUE project_id
DBMS WITH CURSOR proj_cur EXECUTE
```

Figure 29-3 is a sample screen displaying the results.



Project	Employee	Task
1001	Jones	A
	Carducci	A
	Bryant	C
1004	Carducci	B
	Mohr	A
	Silver	B
	Thomas	D
1031	Jones	E

Figure 29-3 The Panther layout is easier to read than the table layout.

Refer to page 11-54 in the *Programming Guide* for more information.

Redirecting Select Results to Other Targets

If you need other destinations for `SELECT` statements, `DBMS CATQUERY` allows you to concatenate a full result row and write it to either a text file or to a Panther variable:

```
DBMS [WITH CURSOR cursor] CATQUERY TO FILENAME file \
    [SEPARATOR "text"] [HEADING [ON | OFF] ]
```

```
DBMS [WITH CURSOR cursor] CATQUERY TO pantherVar \
    [SEPARATOR "text"] [HEADING [ON | OFF] ]
```

There is also a command for formatting the results,

```
DBMS [WITH CURSOR cursor-name] FORMAT [ column ] format
```

For more information on these commands, refer to page 11-10 and to page 11-36 in the *Programming Guide*.

CHAPTER

30 Writing Information to the Database

This chapter discuss how Panther passes data from an application screen to a database. The topics are:

- Colon preprocessing—Using the colon preprocessor to put Panther values into SQL statements. Its forms are `:variable`, `:+variable`, and `:=variable`.
- Parameters—Binding values to SQL parameters when executing a named cursor. The form is `::variable`.

The `DBMS RUN` command is used to execute SQL statements if no data is being returned to Panther, for example, for `UPDATE`, `INSERT`, and `DELETE` statements.

For information on sending data in service requests and calling methods in three-tier applications, refer to Chapter 5, “Defining Services in JetNet and Oracle Tuxedo Applications,” in *JetNet/Oracle Tuxedo Guide*.

Colon Preprocessing

Panther supports colon preprocessing as part of its standard JPL syntax.

Refer to “Colon Preprocessing” on page 19-27 for a description of standard colon preprocessing. One or more colon variables can appear anywhere in a `DBMS` statement; however, The first word in the statement cannot be colon-expanded. Therefore, the following two statements are illegal:

```
:verb SELECT * FROM students
```

```
:command EXECUTE cursor1
```

JPL must know the command word to perform syntax checking and compilation before executing a JPL statement.

In addition to the standard forms of colon preprocessing, Panther's database drivers support special forms of colon preprocessing for values sent to a database. The forms are:

`:+variable`	Colon plus for preprocessing of column values
`:=variable`	Colon equal for preprocessing of operator and column values

These forms of colon preprocessing replace a variable with its value and format it in a style that is appropriate for a column value in an `INSERT` statement, an `UPDATE` statement, or a `WHERE` clause (described below).

Colon-plus Processing

Before colon preprocessing a statement, JPL determines which engine to use. If executing a `DBMS` statement, the JPL parser examines the statement for a `WITH ENGINE` clause. If it finds the clause, it uses the specified engine. If it finds a `WITH CONNECTION` clause, it uses the connection's engine. If neither clause is used, JPL uses the engine of the default connection. Colon-plus processing is not necessary in statements using the `WITH CURSOR` clause. The only `WITH CURSOR` statement that uses column values is `DBMS EXECUTE` and this statement uses binding, not colon-plus processing, to supply column values.

For each `:+variable` used in a JPL statement, the following steps are performed:

Perform Standard Colon Preprocessing

Panther searches for variable in the following places:

- JPL variables local to the procedure that JPL is executing.
- JPL variables local to the module containing the procedure that JPL is executing.
- Widgets on the current screen.
- LDB variables.

When it finds the variable, it copies its value to an internal work buffer. Any formatting is performed on this copy. The variable's contents remained unchanged.

When a screen entry function is executed, it, by default, searches for variable in the LDB before searching the current screen. For more information on variables and their scope, refer to page 19-24, “Variables.”

Determine the Variable's Panther Type

Specific widget properties are examined to determine the variable's Panther type, which in turn, determines how to format the data. Since a variable can have more than one qualifying property specification, Panther uses some precedence rules when assigning a Panther type.

A widget or LDB variable has exactly one Panther type (one of the following):

`DT_BINARY`	`FT_CHAR`	`FT_INT`	`FT_UNSIGNED`
`DT_CURRENCY`	`FT_DOUBLE`	`FT_LONG`	`FT_VARCHAR`
`DT_DATETIME`	`FT_FLOAT`	`FT_PACKED`	`FT_ZONED`
`DT_YESNO`	`FT_HEX`	`FT_SHORT`	

The following properties—Null Field, C Type, Data Formatting, and Keystroke Filter—are examined to determine the Panther type. If the property specifications do not resolve the variable's (including JPL variables) type, it is assigned `FT_CHAR` as the Panther type.

Notes: You can use the `c_type` property to determine a variable's Panther type.

Null Field (null_field)

Panther checks to see if the widget has `null_field` set to `PV_YES`. If the value of the variable equals the null value assignment (specified in the Null

Text (`null_text`) property), the processor replaces the variable's value with the database engine's null string. On most engines, it is the string `NULL`. When a variable's value is null, it is not necessary to determine the Panther type.

If a numeric field is blank or empty, Panther substitutes `NULL` as the column's value for that field, even if the Null Field property is set to No. If the column is specified as `NOT NULL` in the database, the engine returns an error.

Consider a widget having the following Format/Display property settings: `null_field` set to `PV_YES`, `null_text` set to * (a single asterisk), and the repeating property set to `PV_YES`. At runtime, the user enters no data in the field represented by this widget; it is considered null, and Panther displays a repeating string of asterisks (****) as the widget's content. The database driver converts the string **** to `NULL` (that is, the value of the engine's null string) before passing it to the database engine.

On the other hand, if the user enters a text in the widget, the processor proceeds to determine the variable's Panther type from other widget properties.

C Type (`c_type`)

This property's value has the highest priority in determining the variable's Panther type; its value is used to assign a Panther type, unless it is set to `PV_DEFAULT` or `PV_OMIT`.

A newly created widget, one you create with the screen editor, is automatically assigned `PV_DEFAULT` as its C type property value. However, widgets that are database-derived (from the repository), are assigned, via the import process, a C type based on the column's data type. You can also explicitly set a C type.

Be aware of C type property settings that conflict with other properties. For example, if a widget has a C Type setting of `PV_INT` and a Data Formatting setting of `PV_DATE_TIME`, its Panther type is `FT_INT`. The date/time format is enforced for data entry, but Panther's database drivers do not convert the date/time format string into a format the engine would recognize.

The Panther type for the C Type property values are as follows:

C type	**Panther type**
`Char String`	`FT_CHAR`
`Hex Dec`	`FT_HEX`

C type	Panther type
`Int`	`FT_INT`
`Unsigned Int`	`FT_UNSIGNED`
`Short Int`	`FT_SHORT`
`Long Int`	`FT_LONG`
`Float`	`FT_FLOAT`
`Double`	`FT_DOUBLE`
`Zoned Dec`	`FT_ZONED`
`Packed Dec`	`FT_PACKED`

Data Formatting

Checks this property to determine if the widget expects date/time (`PV_DATE_TIME`) or numeric data (`PV_NUMERIC`), determining if the Panther type is `DT_DATETIME` or `DT_CURRENCY`, respectively.

If the Data Formatting property is set `PV_NONE`, Panther examines the Keystroke Filter property.

Keystroke Filter

Checks this property if the variable is neither a date/time or numeric field. If the Keystroke Filter property is set to `PV_DIGITS_ONLY`, the Panther type assignment is `FT_UNSIGNED`; if it is set to `PV_YES_NO` (accepts Yes/No values), the Panther type assignment is `DT_YESNO`; if it is set to `PV_NUMERIC` (accepts numbers only), the type assignment is `FT_DOUBLE`.

Format a Non-null Value

Once a non-null variable's Panther type is determined, this classification is used to perform any necessary formatting before returning the formatted text to JPL.

DT_DATETIME Variables

The processor calls the support routine to format the text in the engine's default syntax for date/time. Some support routines store a Panther format string (defined in the `date_format` property) in the style of the engine. When formatting a field value, it can simply pass the format string and value to Panther's date/time routines to reformat the string. Other support routines can call a conversion function from the DBMS library to perform the task.

The actual result is dependent on the engine. For example, if the value in a date/time field is December 31, 2019 3:05 PM and the current engine is using the ORACLE support routine, Panther formats the date as:

```
TO_DATE('31122019 150500', 'ddmmyyyy hh24miss')
```

If the engine is using the SYBASE support routine, Panther formats the date as:

```
'Dec 31,2019 3:5:0:000PM'
```

Some engines support more than one data type for date-time columns. For more information, refer to the *Database Drivers*.

FT_CHAR Variables

The processor checks if the engine uses quote and escape characters. By default, an engine uses a single quote for `quote_char`, and a single quote for `escape_char`.

The processor first determines the size of the formatted text by adding the length of the unformatted text and the number of embedded `quote_chars` in the text (and for the enclosing quote characters). If it cannot allocate a buffer large enough for the text, the processor returns the `SM_MALLOC` error. If the allocation is successful, the processor writes the formatted text to the buffer. It puts a `quote_char` at the first position in the buffer and, as it copies each character from the source string to the buffer, it compares the character with `quote_char`. If the character equals `quote_char` the processor puts an `escape_char` before the embedded `quote_char`. A final enclosing `quote_char` is put at the end of the text.

For example, Panther formats the value: `Ms. Penelope O'Brien` to

```
'Ms. Penelope O''Brien'
```

Panther formats the value: `Reported record sales for "The Novice's Guide to PC's"` to:

```
'Reported record sales for "The Novice''s Guide to PC''s"'
```

A few engines do not support both single and double quotes within a character string. For engine-specific information, refer to the *Database Drivers*.

FT_HEX Variables

Panther converts the widget's hexadecimal string to a binary format before writing it to the database. The valid hexadecimal string must be an even-length, null-terminated string consisting only of the following letters and numbers: 0-9, A-F, a-f. No character validation on the string is performed on field exit, but if the string cannot be converted, an error occurs when the SQL statement is executed.

For `FT_HEX` data, colon plus and colon equal processing are not available. However, regular colon expansion can be used.

Single line text widgets containing binary data have a maximum size of 127 bytes. To successfully write data longer than 127 bytes, either declare a variable using `DBMS BINARY` or in the screen editor, change the Widget Type to Multitext and set the Word Wrap property to Yes.

FT_NUMERIC and DT_CURRENCY Variables

The processor calls the function `sm_strip_amt_ptr` to strip editing characters from the numerical string. The function strips all non-digit characters (such as dollar signs) except for an optional leading negative sign and a decimal point. The colon preprocessor does not use precision edits when formatting numeric values.

For example, Panther formats the value $500,000.00 as 500000.00. The value (-89.003) as -89.003, and a value of 001-02-0003 as 001020003.

To preserve embedded punctuation in numeric fields, set the widget's C Type property to Char String (`PV_CHAR_STRING`).

For engine-specific information, refer to Database Drivers.

If a widget, defined to accept numeric or currency data, is empty or blank, Panther substitutes `NULL` as the column's value for that widget, even if the Null Field property is set to No. If the corresponding column is specified as `NOT NULL` in the database, the engine returns an error.

Colon-equal Processing

To specify a null value in a search criteria, most engines require the syntax

```
SELECT select_list FROM table WHERE column IS NULL
```

To select rows where a column value is either known or unknown (that is, `NULL`), use the colon-equal processor. For example:

```
DBMS QUERY SELECT * FROM titles \
   WHERE rating_code :=rating_code
```

If the widget named `rating_code` has the following Format/Display property settings: Null Field set to Yes, Null Text set to * (a single asterisk), and the Repeating property set to Yes. Panther formats the widget's data value PG as ‘PG’and executes the `SELECT` statement:

```
SELECT * FROM titles WHERE rating_code = 'PG'
```

It formats the widget’s “null” value, **** (repeating asterisks), as:

```
IS NULL
```

and executes the `SELECT` statement:

```
SELECT * FROM titles WHERE rating_code IS NULL
```

Writing Character String Data to the Database

Consider a widget named `last_name` (from the VideoBiz database) on a screen having the following property settings:

```
   c_type = PV_DEFAULT
   null_field = PV_NO
   data_formatting = PV_NONE
   keystroke_filter = PV_UN FILTERED
```

Therefore, the Panther type is `FT_CHAR`.

If the widget `last_name` contains the text D’Angelo when the following statement is executed:

```
DBMS QUERY SELECT * FROM customers \
   WHERE last_name = :+last_name
```

Panther passes the query:

```
SELECT * FROM customers WHERE last_name = 'D''Angelo'
```

If `last_name` is empty, Panther passes the empty string, not a null string:

```
SELECT * FROM employee WHERE last_name = ''
```

Null conversion is performed only on variables having the null_field property set to PV_YES.

Writing Date/Time or Null Data to the Database

Consider that the widget member_date, from the VideoBiz database, is defined to accept a date (in the form MM/DD/YYYY) and also allow for null data. Given these property settings:

Property/Subproperty	Setting
c_type	PV_DEFAULT
keystroke_filter	PV_DIGIT_ONLY
data_formatting date_format custom_format	PV_DATE_TIME PV_DATE4 MON2/DATE/YR4
null_field	PV_YES
null_text	00/00/0000

The Date/Time setting has a higher precedence than the Keystroke Filter setting, therefore, the Panther type for the widget is DT_DATETIME. If widget contains the date 12/31/2015, and the following function is executed:

```
DBMS WITH CONNECTION oracle_conn RUN \
   INSERT INTO customers (cust_id, last_name member_date) \
   VALUES (:+cust_id, :+last_name, :+member_date)
```

Panther passes the following statement to the engine (in this example, ORACLE is the engine):

```
INSERT INTO customers (cust_id, member_date) VALUES \
   (43, 'D''Angelo', \
    TO_DATE('31122015 000000', 'ddmmyyyy hh24miss')
```

If no date is entered in the member_date field, its content is 00/00/0000 and Panther passes the following statement to the engine:

```
INSERT INTO customers (cust_id, last_name, member_date) \
   VALUES (43, 'D''Angelo', NULL)
```

Writing Numbers as Character Strings to the Database

If a widget accepts only numeric values (`keystroke_filter = PV_DIGITS_ONLY`), such as a telephone number, the colon-plus processor formats the widget's value as an unsigned integer, removing embedded punctuation and leading zeros. However, if the C Type property is set to `PV_CHAR_STRING`, the colon-plus processor formats the widget's contents as a character string, preserving embedded punctuation and leading zeros. The C Type property takes precedence over other property specifications.

For example, if the number 00912 is entered in the `postal_code` widget whose C Type is Char String, and the following statement is executed:

```
DBMS QUERY SELECT * FROM customers \
   WHERE postal_code = :+postal_code
```

Panther passes the following query, submitting the data as a character string, to the engine:

```
SELECT * FROM customers WHERE postal_code = '00912'
```

On the other hand, if the Keystroke Filter property is set to Digits Only, and the C Type is not set to Char String, the following query, using numeric data, is passed to the engine:

```
SELECT * FROM customers WHERE postal_code = 912
```

Writing Hexadecimal Values to the Database

Setting a widget's C Type property to Hex Dec is one method used to fetch binary values to screens. With this setting, when binary data are fetched in a `SQL SELECT` statement, Panther converts the binary value to a hexadecimal string. If any subsequent database updates use the data, it is converted back to a binary format before being passed to the database engine.

Using Parameters in a Cursor Declaration

Some engines permit parameters in the SQL statement of a cursor declaration statement. Therefore, they permit one or more values to be supplied when the cursor is executed. On those engines that do not support binding (for example, SYBASE), Panther internally supports cursors with parameters.

When Panther executes a `DECLARE CURSOR` statement, it scans the statement for parameters. For all engines, Panther recognizes the following syntax to be a parameter:

```
::parameter
```

Many vendors use a single colon to begin a parameter name. Since this form conflicts with the colon preprocessor, two colons must be used in JPL. The second colon prevents the colon processor from performing variable substitution. Some vendors, such as Informix, use a single question mark to represent a parameter. Panther also recognizes these engine-specific forms.

If Panther finds a parameter, it sets up a data structure for it. It attempts to find a value for the parameter when the cursor is executed. Parameters can be used to supply column values for any `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. For example,

```
DBMS DECLARE a_cursor CURSOR FOR \
   SELECT * FROM customers WHERE last_name = ::xyz

DBMS DECLARE b_cursor CURSOR FOR \
   INSERT INTO actors VALUES (::actor_id, ::last_name, \
   ::first_name)

DBMS DECLARE c_cursor CURSOR FOR \
   UPDATE customers SET address1=::address1, \
   address2=::address2, city=::city, \
   state_prov=::state_prov,postal_code=::postal_code \
   WHERE cust_id=::cust_id

DBMS DECLARE d_cursor CURSOR FOR \
   DELETE FROM users WHERE logon_name=::id
```

The binding data structures are stored with an individual cursor. Therefore, the application should give a unique name to each parameter belonging to a single cursor. A cursor cannot have two parameters with the same name.

A value for a parameter is supplied in the USING clause of an EXECUTE statement,

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

Panther looks for the keyword USING before passing the cursor's query to the DBMS. If it finds the keyword, it assumes the arguments which follow are parameter values. If an arg is not quoted, Panther assumes it is a variable and performs variable substitution and formatting. Values and parameters can be bound by position. For example,

```
DBMS DECLARE b_cursor CURSOR FOR \
INSERT INTO roles VALUES (::p1, ::p2, ::p3)
....
DBMS WITH CURSOR b_cursor EXECUTE \
   USING title_id, actor_id, role
```

Values and parameters can be bound explicitly by name:

```
DBMS DECLARE b_cursor CURSOR FOR \
   INSERT INTO roles VALUES (::p1, ::p2, ::p3)
....
DBMS WITH CURSOR b_cursor EXECUTE \
   USING p3=role, p1=title_id, p2=actor_id
```

The preceding declaration, p3, p1, and p2 are not Panther variables but role, title_id, and actor_id are. Panther uses the values of role, title_id, and actor_id to execute the INSERT. To supply a literal value to the INSERT, enclose the value in quotes:

```
DBMS WITH CURSOR b_cursor EXECUTE \
   USING p3=role, p1="89", p2=actor_id
```

Panther formats binding values in a method similar to the colon-plus processor. This is discussed in detail in the next section.

On those engines that support parameters, using them can improve the efficiency of the application, especially when a cursor is executed several times. On engines where Panther simulates support, such as SYBASE, the use of parameters is less efficient. However, the convenience and the greater ease of portability can compensate for the additional processing.

Parameter Substitution and Formatting

An arg in a USING clause can be:

- A quoted string
- A Panther variable

Colon-plus processing is not necessary because Panther automatically formats the value of parameter variables. If the variable is an array, an occurrence number can be given. If no occurrence is given, Panther concatenates all the non-empty occurrences in the array, separating the occurrences with a single space. Substrings are not permitted.

For each cursor, Panther maintains binding information. When a cursor's statement uses parameters, Panther stores the names of the parameters. When a cursor is executed, Panther compares the values in the `DBMS EXECUTE` statement with the binding information from the cursor's declaration. This permits both positional and explicit binding.

Panther uses a data structure to store the formatted text and Panther type of `arg`. If `arg` is not quoted, Panther assumes it is a variable and determines the variable's data type. Like the colon-plus processor, the binding routine distinguishes between empty and null variables; a variable is null if it the Null Field (`null_field`) property is set to `PV_YES` and the variable contains the null string.

If the Panther type is `DT_DATETIME`, Panther calls the support routine to convert the value to a binary date/time value. For more information, refer to page 30-9, “Writing Date/Time or Null Data to the Database.”

No processing is done on the values of `FT_CHAR` variables or quoted strings.

For all other types, Panther strips characters other than digits, the decimal point, and a leading negative sign from the value.

The following examples show the different formats for arg in a USING clause:

```
DBMS DECLARE x CURSOR FOR \
   SELECT * FROM titles \
   WHERE title_id=::p1 OR genre_code=::p2

# newid and newtype are LDB variables
DBMS WITH CURSOR x EXECUTE \
   USING p1=newid, p2=newtype

# For p1, a literal value is supplied.
# For p2, code is a JPL variable with the initial text
# "film_type." film_type is also a widget on the current
# screen and this widget supplies the parameter's value.
DBMS WITH CURSOR x EXECUTE USING p1='92', p2=:code
```

```
# id and vid_type are field arrays. i is a JPL variable
DBMS WITH CURSOR x EXECUTE \
   USING p1=id[i], p2=vid_type[i]
```

Writing Currency Values to the Database

Consider a widget, `rent_amount` from the videobiz database, with the following property settings: `c_type = PV_DEFAULT`, `keystroke_filter = PV_NUMERIC`, `data_formatting = PV_NUMERIC`, and `numeric_type = PV_DEFAULT_0` (Format Type is set to Local Currency). Its Panther type is `DT_CURRENCY`.

If the data entered in the widget is `$1,000.99`, and the following statements are executed:

```
DBMS DECLARE sales_cursor CURSOR FOR \
   SELECT * FROM customers WHERE rent_amount > ::x
...
DBMS WITH CURSOR sales_cursor EXECUTE USING x=rent_amount
```

the engine executes:

```
SELECT * FROM customers WHERE rent_amount > 1000.99
```

Writing Data from Arrays

Consider the widget named notes, a multiline text widget, with its Word Wrap property set to Yes and with the following property settings:

Property/Subproperty	Setting
`c_type`	`PV_DEFAULT`
`keystroke_filter`	`PV_UNFILTERED`
`data_formatting`	`PV_NONE`
`null_field`	`PV_YES`
`null_text`	`no specification`

This widget is a Panther type `FT_CHAR`. If you execute the following statements:

```
DBMS DECLARE ins_cursor CURSOR FOR \
   INSERT INTO customers (cust_id, notes) \
   VALUES (::p1, ::p2)...
DBMS WITH CURSOR ins_cursor EXECUTE USING cust_id, notes
```

when the widget is empty, the DBMS executes:

```
INSERT INTO customers (cust_id, notes) VALUES (123, '')
```

If, however, the widget contains text, Panther concatenates the non-empty occurrences into one long string which the DBMS inserts into the column notes.

```
INSERT INTO customers (cust_id, notes) \
   VALUES (123, 'This customer wants to be notified \
when A River Runs Through It is available for rental.')
```

Notes: For multiline text widgets, the Max Occurrences property should also be set to avoid memory allocation errors.

CHAPTER

31 Building a Transaction Manager Screen

The screen wizard is the easiest way to build screens that use the transaction manager. The wizard leads you through choosing your root table view, selecting widgets from that table view, adding additional table views and their widgets, and it automatically changes the property settings.

Even so, if you want to modify screens created by the screen wizard or to build your own screens or service components that use the transaction manager, you need to understand how the transaction manager uses property settings, links and table views in order to automatically generate SQL statements and to perform database processing. This chapter summarizes how you can build a two-tier application screen in the editor.

To help explain the transaction manager concepts, a sample screen based on the videobiz database is used. The screen lets you enter a video title by name or identification code, and view the names of actors and their roles in that video. A picture of the screen appears on page 31-4 as Figure 31-1.

Development Process for Transaction Manger

Database operations are integrated into the overall development of your application. Table 31-1 shows how database operations are effected by each step in developing a transaction manager application.

Table 31-1 Application development process and the effect on database operations

Step	Effect on database operations
Import database tables into the repository.	Repository information is used at all stages of screen design and database operation development.
Create a screen.	Using the screen wizard: widgets, table views, and links are copied from the repository to the screen being generated. Default properties are set. Default transaction manager commands are included.
	Using the editor: copy widgets and links from the repository to the screen being created. Table views will automatically be copied. Link properties will need to be updated as part of the next step.
Edit table views and links.	Changes transaction manager processing. For example, you can change the order in which table views are processed, or make a table view non-updatable.
Edit widget database properties.	Changes SQL generation. For example, you can exclude a column from `SQL SELECT` statements or specify a validation link.
Create push buttons or menu items (on non-wizard screens).	Calls to transaction manager commands.

Table 31-1 Application development process and the effect on database operations

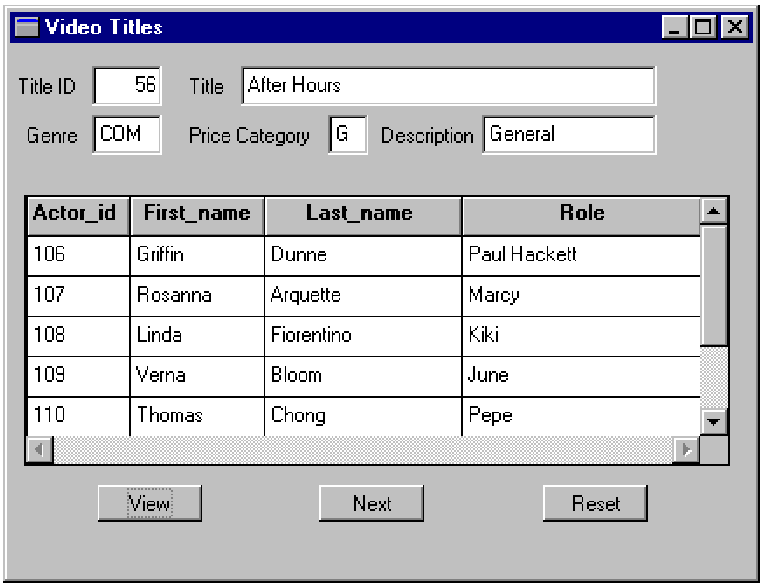
Step	Effect on database operations
Write event functions.	Invokes the transaction manger or database interface to perform database operations.
Edit transaction styles and class.	Changes the behavior of data entry widgets on your screen for specified operations.
Assign a transaction event function or a transaction model to a table view.	Changes transaction manager processing for the specified table view.
Assign a screen-level transaction model or edit a database-specific model.	Changes transaction manager processing for all table views in a screen or application.

Copying Repository Objects

Building the sample screen was accomplished by using the database importer and the repository. The videobiz database was created in JDB and imported into the repository. The database importer created a repository entry for each database table. The name of the repository entry corresponds to the database table name so that entries can be easily identified. Since objects copied from the repository inherit property settings, the application screen contains much of the information needed for SQL generation and database access automatically.

Each repository entry contains:

- A widget corresponding to each database column. (One of the properties for the widget is the column name.)
- A label for each database column.
- A table view containing database table information.
- Links based on any foreign key definitions for the database table.



Title ID: 56
Title: After Hours
Genre: COM
Price Category: G
Description: General

Actor_id	First_name	Last_name	Role
106	Griffin	Dunne	Paul Hackett
107	Rosanna	Arquette	Marcy
108	Linda	Fiorentino	Kiki
109	Verna	Bloom	June
110	Thomas	Chong	Pepe

Figure 31-1 Sample screen used to explain transaction manager processing.

Sequence for Copying Objects

When you create a screen with multiple table views, the order used to copy objects from the repository is important. You need to copy the information for the major table views in the screen first. This ensures that any primary key widgets copied to the screen are in the master, or parent, table view.

Since the focus of the sample screen is information about each video title, widgets and links from the titles repository entry are copied first.

Table 31-2 Objects copied from the titles repository entry

Repository entry	Type of Widget	Name
`titles`	Text	`title_id`, `name`, `genre_code`, `pricecat`
	Labels	
	Table View	`titles` (copied automatically with the text widgets)
	Link	`K1titles` (`pricecats+titles`)

Actor information is in two different database tables, `actors` and `roles`. The `roles` table with its a `title_id` column and link to the titles table view must be copied next. (Note that it is not necessary to copy the `title_id` widget itself from the roles entry; the transaction manager uses the `title_id` widget in the titles table view for SQL generation.) Just the `actor_id` and the `role` widgets are copied to the screen—`actor_id` because it is part of the primary key.

Table 31-3 Objects copied from the roles repository entry

Repository Entry	Type of Widget	Name
`roles`	Text	`actor_id`, `role`
	Table View	`roles` (copied automatically with the text widgets)
	Link	`K1roles` (`titles+roles`) `K2roles` (`actors+roles`)

Since `actor_id` is already onscreen, all that is needed from the actors entry are `first_name` and `last_name`.

Table 31-4 Objects copied from the actors repository entry

Repository Entry	Type of Widget	Name
`actors`	Text	`first_name`, `last_name`
	Table View	`actors` (copied automatically with the text widgets)

Finally, since the price category codes are not self-explanatory, `pricecat_dscr` is copied from pricecats to provide better descriptions.

Table 31-5 Objects copied from the pricecats repository entry

Repository Entry	Type of Widget	Name
`pricecats`	Text	`pricecat_dscr`
	Table View	`pricecats` (copied automatically with the text widget)

Specifying the Traversal Order

Since the screen contains widgets from several database tables, you must specify in which order the tables will be processed. Copying the objects from the repository in the correct sequence is the first step. The next step is to modify table view and link properties.

Table Views

A table view is a group of related widgets, generally belonging to the same database table. Table view properties include the name of the database table, the names of columns that belong to the database table, and the columns that comprise the table's primary key. The import process creates a table view, an invisible widget type, for each imported table, and includes the widgets corresponding to the database columns in each table view.

If a widget is a member of a table view in the repository, Panther automatically adds the widget to a table view of the same name in the destination. If the table view does not exist, Panther creates it using the properties of the table view in the repository. Thus, most table views are created automatically by the database importer and copied from the repository as the widgets are copied.

However, just knowing the table views on a screen does not tell the transaction manager which table view should be processed first. To obtain this information, the transaction manager looks at the link properties for the screen.

Links

A link defines the relationship between two table views. The link properties list the columns or widgets connecting the two table views, the type of link—server or sequential, and the parent and child table view designations.

The import process creates links based upon foreign key information contained in the database. If the database contains no foreign key information, then you can create the links manually in the screen editor. Link widgets are only visible in the screen editor workspace, not at runtime.

How to Gain Access to Table View and Link Properties

You can select the table view or link either from the Widget List or the DB Interactions window.

How to View the Table Views and Links for a Screen

Choose View→DB Interactions. The DB Interactions window provides a graphical representation of the table viers and links on your screen. You can also click on any of the objects in the window in order to gain access to its properties.

Setting Link Properties

Determining the Root Table View

The table view listed at the top of the DB Interactions screen is the root table view, the first table view to process for this screen. The transaction manager determines the root table view from the Parent and Child properties of all of the links on a screen. Since the purpose of the sample screen is to provide information about each video title, `titles` is the root table view, as illustrated in Figure 31-2 (page 31-10).

If you get an error message that the root table view cannot be determined, check the Parent and Child properties for the link. Often, these properties need to be reversed for one or more links. If changing these properties does not resolve the error, you can set the root table view manually in the screen properties.

Determining the Order of Processing

In a link widget, two table views are identified: one table view is designated as the parent table view and the other is designated as the child. This designation helps determine the root table view and the order of processing for the table views.

Generally, the database table associated with the Parent table view is different than the one associated with the Child table view. One exception to this condition is for SQL self-joins where the same database table (using different table view names) is associated with both the Parent and Child table views.

When you copy links from the repository, the settings for the Parent and Child properties might need to be reversed for a particular screen. You can easily determine the current values by looking at the link in the editor. The link is displayed as the parent table view name plus (+) the child table view name.

In the sample screen, some of the Parent and Child properties had to be edited. Since `titles` is the root table view, it must be the parent table view for any link in which it appears. Since the `K1titles` link had `titles` as the child table view, the Parent and Child properties of that link were changed for this screen. `titles` became the Parent and `pricecats` the Child.

At the next level, `roles` needs to be the parent table view for any link in which it appears. For the `K2roles` link, `roles` became the Parent, and `actors` the Child.

Notes: When you reverse the Parent and Child settings, you must also edit the Relations property if the columns joining the two tables do not have the same name. This was not needed for our sample screen since the `pricecat` column in the titles table has the same name as the `pricecat` column in the `pricecats` table.

Restrictions

You cannot have a cycle appearing in the link specifications. For example, if `link1` declares the titles table to be the parent and the roles table to be the child, `link2` cannot have the roles table be the parent and titles be the child. That constitutes a circular link. Remember, however, that links are specific to one screen. On another screen, the relationship specified in `link2` could exist.

You cannot have the same table view in both the Parent and Child properties. If this occurs, the error message "Maximum depth exceeded" is displayed.

Specifying the Link Type

There are two types of links:

- Sequential link represents a one-to-many relationship between two table views.
- Server link represents a one-to-one relationship between two table views.

A group of table views connects by server links is referred to as a server view. Therefore, a table view can also be a server view which means that it is either the root table view or is connected to a parent table view by a server link.

The transaction manager uses server views to process your database operations more efficiently. When the transaction manager retrieves data from the database (invoking the SQL `SELECT` command), it processes all of the table views in a server view as a single SQL `SELECT`, rather than repeating the `SELECT` once for each table view. Other database operations (such as those that update the database) are processed differently; refer to page 36-7 for more information.

In the sample screen, a sequential link between the `titles` and `roles` table views was appropriate, but the Type property had to be updated to Server for the links between the `titles` and `pricecats` table views and the `roles` and `actors` table views.

Tree Traversal

The DB Interactions screen graphically illustrates the tree of table views and links that the transaction manager uses to perform its processing. On this screen, < (less than) designates a sequential link; = (equals) designates a server link.

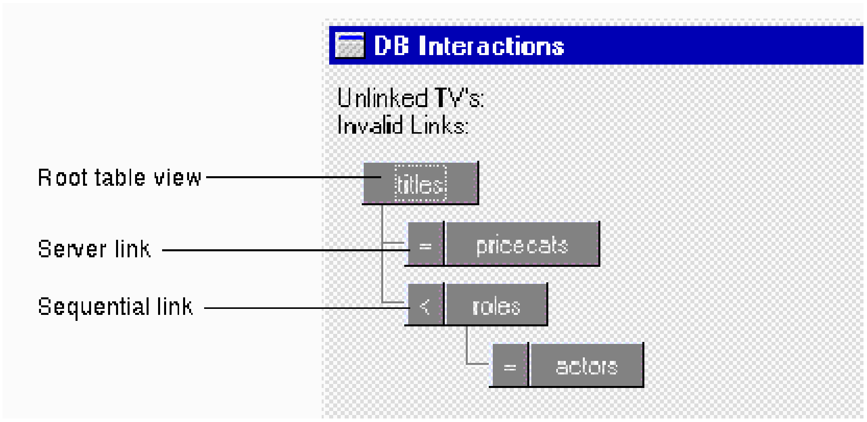


Figure 31-2 DB Interactions screen for the sample screen showing the linked table views and the link type.

When a command is selected, the transaction manager traverses this table view tree, issuing statements to each table view, or server view, in order to fetch or update data in the database.

In our sample screen, there are two server views:

- `titles` (which includes the `pricecats` table view)
- `roles` (which includes the `actors` table view)

For server links, you can specify the Join Type—the join operation the `SELECT` statement will use to combine information from the database tables. In order to be available, the database engine must support outer joins. The Join Type property can be set to: Inner (default), Left Outer, Right Outer, or Full Outer. For more information on how the Join Type affects data retrieval, refer to page 33-28.

Setting Table View Properties

For a widget to participate in transaction manager processing and SQL generation, the widget must be a member of a table view. In addition, if members of the table view participate in `SQL INSERT`, `UPDATE` and `DELETE` statements, all members must have the same number of onscreen occurrences and the same number of maximum occurrences.

Typically, widgets are automatically assigned to a table view when you copy them from the repository to your screen. You can add members to an existing table view in order to:

- Include a widget in the SQL generation.
- Create a “virtual” column—that is, use a widget on the screen to display information from the database even though the widget does not have an associated column in the database.
- Add a database column back to the table view that you previously removed from the table view.

Refer to page 22-10 in the *Using the Editors* for details on how to define a widget's membership in a table view.

You can specify whether a widget or a table view participates in SQL generation and database updates. For table views, the Updatable property determines if data in the corresponding table can be updated. If set to No, widgets in that table view are protected from focus and data entry.

Specifying Widget Properties

Once the screen contains the necessary widgets, table views and links, you might choose to edit some of the Database or Transaction properties. Editing the properties can change the transaction manager processing or SQL generation performed for commands.

Changing SQL Generation

For the sample screen, properties were changed for the `title_id` and name widgets in order to fetch a specific video title.

Table 31-6 Property Changes

Widget name	Database - Fetch Data category
`title_id`	Use In Where property set to Yes
	Operator set to =
`name`	Use In Where property set to Yes
	Operator set to %link%

With these changes, when a user enters an identification code or part of a video title followed by the `SELECT` or `VIEW` commands, the transaction manager will display the desired information. For more information on setting properties for SQL generation, refer to page 33-1.

Using Grids

For the grid widget containing the actor information, the following Format/Display properties were changed:

- The Column Titles property was changed to Per Column in order to display the column name set by the importer in the widget's Column Title property.
- The Row Titles property was set to None.

Using Validation Links

The links that are defined for a screen can also be used to specify validation links. When a validation link exists, you can enter a value in a widget, in either new or update mode, and the transaction manager looks up that value in the linked database table. If the value exists, it displays data for any widgets in the child table view. If the value does not exist, it displays the error Invalid Entry.

It is very simple to specify a validation link. Create the desired link if it does not exist. Then, set the Validation Link property for the widget containing the entered value to that link.

To view a sample validation link, refer to page 33-46.

Specifying Transaction Manager Commands

Transaction manager processing is implemented by invoking transaction manager commands. In the sample screen, the commands are called via control strings on the screen's push buttons. When a transaction manager command is activated at runtime, the events associated with that command are generated, which in turn perform the processing needed.

On the sample screen, the push buttons are associated with the following commands:

- View—The `VIEW` command generates the `SQL SELECT` statements necessary to display data for all table views. If a video name or id is entered, the SQL generation is modified to display data for that entry only.

Control String property: ^`sm_tm_command`(“`VIEW`”)

- Next—The `CONTINUE` command displays the next title and all its associated data.

  Control String property: ^`sm_tm_command`(“`CONTINUE`”)

- Reset—Two transaction manager commands are used to reset the screen. The `CLEAR` command clears data in all the screen's widgets; the `CLOSE` command resets the screen to initial mode in order to allow user input for the video name or id.

  Control String property:

  `^(^sm_tm_command(“CLEAR”))sm_tm_command(“CLOSE”)`

Refer to page 31-13 for information on specifying transaction manager commands.

Refer to page 35-2 for information on how the transaction manager generates events for each command.

Refer to page 33-43 for information on viewing the SQL that the transaction manager generates for each command.

Changing the Transaction Mode

Not only does calling a command generate the events for that command, it also sets the transaction mode for the screen. The transaction mode helps determine whether fields are protected from data entry and if push buttons and menu items are active.

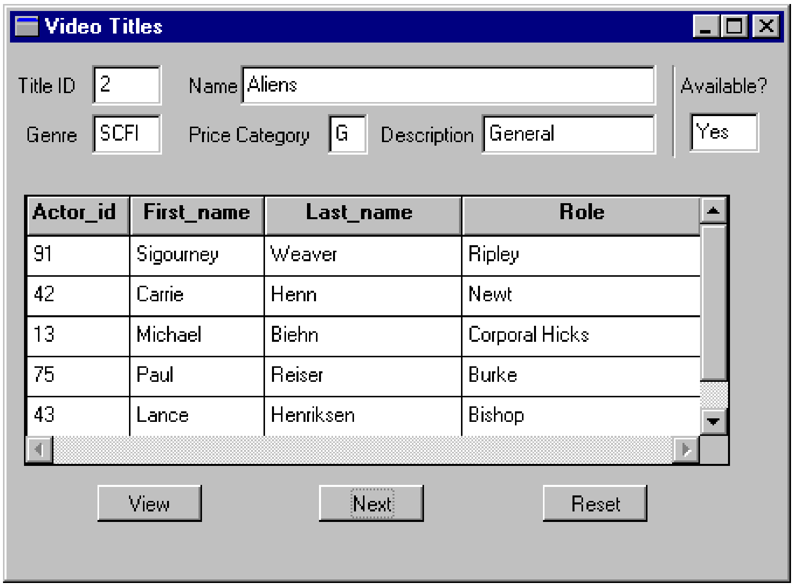
The behavior of widgets in the current transaction mode is determined by the Class property setting. The push buttons were assigned the following values in order to make them inactive for certain modes:

- View—`view_button`—active in initial and view mode
- Next—`continue_view_button`—active in view mode
- Reset—`clear_button`—active in all modes

Refer to page 36-12 for more information on transaction modes and Class property settings.

Adding a Transaction Event Function

Part of the transaction manager's power is that it allows you, the developer, to modify its processing as needed. One way of customizing the transaction manager is to write an transaction event function and modify or replace the default processing for a transaction manager event. In our sample screen, an additional field was desired, stating whether a copy of the video is available.



Title ID	2	Name	Aliens	Available?	
Genre	SCFI	Price Category	G	Description	General
				Available?	Yes

Actor_id	First_name	Last_name	Role
91	Sigourney	Weaver	Ripley
42	Carrie	Henn	Newt
13	Michael	Biehn	Corporal Hicks
75	Paul	Reiser	Burke
43	Lance	Henriksen	Bishop

Figure 31-3 An additional field, processed by a transaction manager event function, displays whether the video is available for rental.

In order to display this information, the following steps were taken:

- From the tapes repository entry, the `title_id` field and the `titles+tapes` link were copied to the sample screen. This automatically created the tapes table view.

- The `title_id` widget was renamed `title_avail`, since each widget's name on the screen must be unique.
- For the `title_avail` widget, the Use in Select property (under Database→Fetch Data) was set to No.

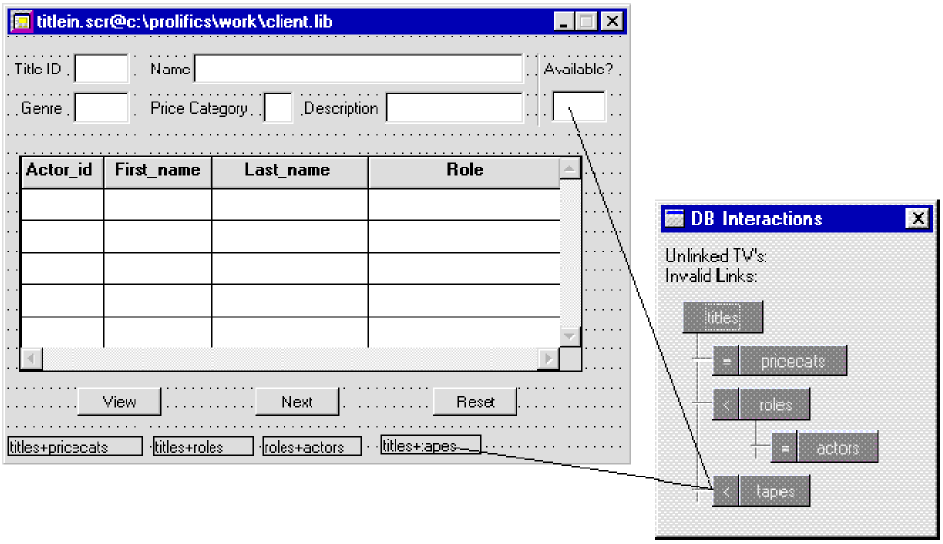


Figure 31-4 The field named title_avail, which also corresponds to the title_id widget, contains the results of the transaction manager event function.

- Under JPL Procedures, the following event function was added for the main request of the `VIEW` command:

```
// TM event functions are passed one parameter:
// the event name.

proc find_tapes (event)

// The VIEW command's major request is TM_VIEW.

if event == TM_VIEW
{
// Create a JPL variable and alias it to receive
// the value from the subsequent SELECT statement.

vars numtitles
DBMS ALIAS numtitles
DBMS QUERY SELECT count (distinct title_id) \
FROM tapes \
WHERE title_id = :+title_id and status = 'A'

// Update the title_avail field based on the value.

if numtitles>0
{
call sm_n_putfield("title_avail", "Yes")
}
else
{
call sm_n_putfield("title_avail", "No")
}

// Event processing is self-contained. No further
// processing is needed.

return TM_OK
}
return
```

- The event function was attached to the tapes table view in its Function property.

Refer to page 32-1 for more information on writing transaction manager event functions.

Refer to page 36-1 for information about transaction manager processing at runtime.

CHAPTER

32 Writing Transaction Event Functions

A transaction manager event function replaces part of the functionality provided by a transaction model. You can write transaction event functions to:

- Modify generated SQL.
- Supply hand-coded SQL or stored procedure calls to replace generated SQL.
- Modify or query properties using the property API.
- Change error handling.

In order to use a transaction manager event function, you need to:

- Write a function which includes:
  - The event whose functionality you want to modify or replace.
  - Processing to be added to the event.
  - Return codes which tells the transaction manager how to proceed.
- Install the event function by:
  - Setting the table view's Function property to the name of the JPL module containing the function.
  - Modifying the transaction model.

This section explains how to write a simple event function, how to specify the appropriate return code, how to modify processing for SQL `SELECT` statements, and how to modify processing for SQL `INSERT`, `UPDATE` and `DELETE` operations.

The Nature of TM Event Functions

Transaction manager event functions can allow you to replace Panther's automatically generated SQL with your own, hand-coded SQL. The following example, a simple event function, executes a custom SQL statement:

```
proc simpleEventFunc (event)
if event == TM_VIEW || event == TM_SELECT
{
   DBMS QUERY \
      SELECT title_id, name, genre_code
   call sm_tm_iset(TM_OCC_COUNT, @dmrowcount)
   return TM_CHECK
}
return TM_PROCEED
```

To invoke the event function, the `function` property on the `titles` table view is set to `simpleEventFunc`. The function is called each time an event is processed for this table view. You can have an event function for each table view; however, for database queries, the event function must be specified in the first table view of a server view.

Since transaction event functions are passed one argument, specifically the transaction event, you need to identify which events you want to modify. In the `simpleEventFunc` function, the SQL statement replaces the processing for `TM_VIEW` or `TM_SELECT` events. For all other events, the return code `TM_PROCEED` tells the transaction manager to go ahead and call the transaction model, as if the event function had not been called, so that the transaction model's processing is performed.

However, the `simpleEventFunc` function specifies that if the event is `TM_VIEW` or `TM_SELECT`, the transaction manager uses the database driver's `DBMS QUERY` command to retrieve data from `titles` table. Calling `sm_tm_iset` sets the number of rows returned for this server view. The return code `TM_CHECK` tells the transaction manager to test for an error; the transaction manager can check for database errors from the `SELECT` statement.

Specifying a Return Code

Specifying the correct return code at the end of an event function tells the transaction manager what to do next. Table 32-1 summarizes possible return values.

Table 32-1 Return codes used by the transaction manager

Return value	Means
`TM_CHECK`	Test to determine if an error occurred. (Used in data base-specific transaction models to check for SQL execution errors.)
`TM_CHECK_ONE_ROW`	Test to determine if an error occurred; event processing is considered successful only if exactly one row was affected.
`TM_CHECK_SOME_ROWS`	Test to determine if an error occurred; event processing is considered successful only if one or more rows were affected by the processing.
`TM_FAILURE`	Event processing failed.
`TM_OK`	Event processing succeeded. Further processing of the event is skipped.
`TM_PROCEED`	Proceed to the next step; after completing the event function, proceed as if the function was never called. In the case of a transaction manager event function, this means that the transaction model is called.
`TM_UNSUPPORTED`	Event was not recognized.

Specifying TM_PROCEED

`TM_PROCEED` tells the transaction manager to go ahead and call the transaction model as if the event function had not been called. In the following code example, the transaction model's processing is performed after the `DBMS QUERY` statement is executed.

```
if event == TM_VIEW || event == TM_SELECT
{
    DBMS QUERY \
        SELECT title_id, name, genre_code FROM titles
    call sm_tm_iset(TM_OCC_COUNT, @dmrowcount)
    return TM_PROCEED
}
```

The processing for these requests in the transaction models creates a select cursor, builds the structures to bring back data for all the members of the table view whose Use In Select (`use_in_select`) property is set to Yes, generates the SQL for the `SELECT` statement, and then executes the statement.

The processing for all the events takes place quickly, so you might not visually notice that both sets of processing are being performed unless you view the processing in the debugger and break on each transaction manager event.

If both an event function and a transaction model are called for a single event by using `TM_PROCEED`, the processing in the event function takes place *before* the processing in the model.

Specifying TM_OK

`TM_OK` is used when the processing is contained in itself; therefore, the transaction model is not required for processing and no checking for database errors is required. Since the following statement does not require any processing in the model, the `TM_OK` return code is a possible setting.

```
if event == TM_VIEW || event == TM_SELECT
{
    DBMS QUERY \
        SELECT title_id, name, genre_code FROM titles
    call sm_tm_iset(TM_OCC_COUNT, @dmrowcount)
    return TM_OK
}
```

Checking for Database Errors

Even though `TM_OK` does not check for database errors, the transaction models set `TM_STATUS` to `-1` whenever the database error handler returns an error. If the statement causes the database error handler to be invoked, for example by specifying an invalid column name, the database error handler returns an error. Then, the transaction manager returns an error that the transaction model or event function had an error.

Three return codes are designed specifically for checking for errors from the database engine and from Panther's database drivers: `TM_CHECK`, `TM_CHECK_ONE_ROW`, and `TM_CHECK_SOME_ROWS`.

- `TM_CHECK`—Used to check for any error from Panther's database drivers. If one is found, the transaction manager displays a message to that effect. The event `TM_TEST_ERRORS` is pushed onto the event stack for this return code. This is the best return code for the sample statement.

  ```
  if event == TM_VIEW || event == TM_SELECT
  {
  DBMS QUERY \
      SELECT title_id, name, genre_code \
      FROM titles WHERE title_id = :+title_id
  call sm_tm_iset(TM_OCC_COUNT, @dmrowcount)
  return TM_CHECK
  ```

- `TM_CHECK_ONE_ROW`—Pushes the `TM_TEST_ONE_ROW` event onto the event stack. This event checks the value of `@dmrowcount` (global variable) to make sure its value is equal to `1`. This return code is used following SQL statements that modify the database to make sure that only one row was affected.

- `TM_CHECK_SOME_ROWS`—Pushes the `TM_TEST_SOME_ROW` event onto the event stack. This event checks the value of `@dmrowcount` to make sure its value is greater than or equal to `1`. This can be used to make sure that SQL `SELECT` statements return rows from the database.

Specifying TM_FAILURE

`TM_FAILURE` pushes the `TM_NOTE_FAILURE` event onto the event stack. The type of error message displayed by the transaction manager depends on the value of various transaction manager variables, like `TM_STATUS` or `TM_EMSG_USED`. For more information, refer to page 36-27, "Processing Errors in the Transaction Manager."

Performing Error Checking

In the following event function, `checkingEvent`, the variant does its own checking for errors. It is only checking the `@dmretcode`, which would also be checked when `TM_CHECK` is returned. When checking for errors from database access, it is always important to check `@dmretcode`.

```
proc checkingEvent( event )
if event == TM_SEL_BUILD_PERFORM
{
   DBMS DECLARE :@tm_sel_cursor CURSOR FOR \
      SELECT actor_id, first_name, last_name \
      FROM actors \
      WHERE actor_id = ::actor_id
```

```
    DBMS WITH CURSOR :@tm_sel_cursor EXECUTE USING actor_id
    if @dmretcode != 0 return TM_FAILURE
    call sm_tm_iset(TM_OCC_COUNT, @dmrowcount)
    return TM_OK
}
return TM_PROCEED
```

Unsupported Events

The database-specific transaction models use `TM_UNSUPPORTED` to indicate that the event is not supported in the transaction model. This is important to note in case you add transaction events to the model. This return code pushes the `TM_NOTE_UNSUPPORTED` event onto the event stack.

Modifying SELECT Statement Processing

The following slices are generated for `TM_VIEW` and `TM_SELECT` events:

- `TM_GET_SEL_CURSOR`—Allocates a Panther cursor for use by the `SELECT` statement. Depending on the database, a Panther cursor may or may not correspond to a database cursor.
- `TM_PREPARE_CONTINUE`—Checks the value of the Fetch Directions property for the table view.
- `TM_SEL_GEN`—Generates data structures that are used to build the SQL statements needed to view the data. This slice and the next slice are separated to enable "tweaking" of the SQL that is about to be built.
- `TM_SEL_BUILD_PERFORM`—Builds and executes the SQL statements needed to view the data. Uses the Panther cursor allocated in the `TM_GET_SEL_CURSOR` slice event.
- `TM_SEL_CHECK`—Check to determine whether to give up the Panther cursor allocated in the `TM_GET_SEL_CURSOR` step. This cursor is given up here only if the select set is exhausted.

Replacing a SQL SELECT Statement

The following event function, `myEvent`, uses the transaction model's cursor management and error reporting capabilities. The variable `@tm_sel_cursor` contains the name of the cursor to be used to execute the SQL `SELECT` statement.

```
proc myEvent (event)
if event == TM_SEL_BUILD_PERFORM
{
   DBMS DECLARE :@tm_sel_cursor CURSOR FOR \
      SELECT title_id, name, genre_code \
      FROM titles \
      WHERE title_id = ::title_id
   DBMS WITH CURSOR :@tm_sel_cursor EXECUTE USING title_id
   call sm_tm_iset(TM_OCC_COUNT, @dmrowcount)
   return TM_CHECK
}
return TM_PROCEED
```

The event function is called for the `TM_SEL_BUILD_PERFORM` event. This event builds and executes the SQL statements. Since this slice is generated for both `TM_SELECT` and `TM_VIEW` requests, the event function is called for either request.

In some cases, you might want to check for errors by writing error checking code within the event function. In that case, you should return `TM_FAILURE` if an error is encountered, and `TM_OK` if there is no error.

The `myEvent` function permits the `TM_SEL_GEN` slice to be handled by the transaction model, which (in the case of the database-specific transaction models) has Panther build unneeded data structures. For slightly better performance, that slice could be skipped as follows:

```
proc fasterEvent (event)
if event == TM_SEL_BUILD_PERFORM
{
   DBMS DECLARE :@tm_sel_cursor CURSOR FOR \
      SELECT title_id, name, genre_code \
      FROM titles \
      WHERE title_id = ::title_id
   DBMS WITH CURSOR :@tm_sel_cursor EXECUTE USING title_id
   call sm_tm_iset(TM_OCC_COUNT, @dmrowcount)
   return TM_CHECK
}
if event == TM_SEL_GEN
   return TM_OK
return TM_PROCEED
```

Modifying SQL Generation

In addition to writing event functions which replace the SQL `SELECT` statement, you can also write event functions to modify the automatic SQL generation. Use one or more of the following C functions that are prototyped in `tmusubs.h`:

- `dm_gen_change_execute_using`—Modifies the bind parameters in the `EXECUTE USING` statement.
- `dm_gen_change_select_from`—Modifies the table list in a SQL `SELECT` statement.
- `dm_gen_change_select_group_by`—Modifies the `GROUP BY` clause in a SQL `SELECT` statement.
- `dm_gen_change_select_having`—Modifies the `HAVING` clause in a SQL `SELECT` statement.
- `dm_gen_change_select_list`—Modifies the select list in a SQL `SELECT` statement.
- `dm_gen_change_select_order_by`—Modifies the `ORDER BY` clause in a SQL `SELECT` statement.
- `dm_gen_change_select_suffix`—Adds the specified text to the end of a SQL `SELECT` statement.
- `dm_gen_change_select_where`—Modifies the `WHERE` clause in a SQL `SELECT` statement.

A sample event function which adds a column and its corresponding table to the `SELECT` statement is shown below:

```
proc titlesEvent(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)

   if (retval != 0)
   return TM_FAILURE
```

```
    }
return TM_PROCEED
```

Replacing Other SQL Statements

SQL `INSERT`, `UPDATE`, and `DELETE` statements are generated as part of the processing of the transaction manager `SAVE` command, but only if data is modified or new data is entered. In total, the transaction manager can generate up to six types of requests when processing a `SAVE` command. They are generated in the following order:

- `TM_PRE_SAVE`—Indicates that save processing has started.
- `TM_SAVE`—The transaction models do nothing.
- `TM_DELETE`—Transaction models generate SQL `DELETE` statements for records to be deleted; one per modified record.
- `TM_UPDATE`—Transaction models generate SQL `UPDATE` statements for records to be updated; one per modified record. Note that changing the primary key is implemented by deleting the record with the old value and inserting a record with the new value.
- `TM_INSERT`—Transaction models generate SQL `INSERT` statements for records to be updated; one per entered record.
- `TM_POST_SAVE`—Transaction models do commit and rollback processing here. Rollback processing occurs if there was an error in the saving process. Commit processing occurs only for full implementations of the `SAVE` command, not partials.

`TM_DELETE` is generated before `TM_UPDATE` and `TM_INSERT` in order to prevent duplicate record errors. This is also why `TM_UPDATE` is generated before `TM_INSERT`. Also, a single cursor is used for all `SAVE` operations. This permits all `SAVE` operations to be part of the same database transaction.

The transaction models further slice the `TM_DELETE`, `TM_UPDATE`, and `TM_INSERT` requests. However, the slicing is performed only if slices are needed. For example, when a row is inserted, no slices are generated for the `TM_DELETE` request. The three slices for each of these requests are:

- `TM_GET_SAVE_CURSOR`—Generated only if this is the first `TM_INSERT`, `TM_UPDATE`, or `TM_DELETE` event for the `SAVE` command. Allocate a cursor for use as the save cursor, and, if needed by the database, begin a database transaction.

- `TM_*request*_DECLARE`—Generates the SQL statement and uses the generated statement in the declaration of the cursor. The processing of this slice avoids cursor re-declaration if the proper SQL statement is already declared.
- `TM_*request*_EXEC`—Executes the declared cursor.

Supplying custom `INSERT`, `UPDATE`, and `DELETE` statements should normally be done in the `TM_*request*_DECLARE` events, since they occur only when a new cursor must be declared (which can be a somewhat expensive operation, depending on the database).

The following event function provides custom SQL `INSERT`, `UPDATE`, and `DELETE` statements:

```
proc saveEvent( event )
if ( event == TM_DELETE_DECLARE )
{
   DBMS DECLARE :@tm_save_cursor CURSOR FOR \
      DELETE FROM actors WHERE actor_id=::w_actor_id
   return TM_CHECK
}
if ( event == TM_UPDATE_DECLARE )
{
   DBMS DECLARE :@tm_save_cursor CURSOR FOR \
      UPDATE actors SET first_name=::s_first_name, \
      last_name=::s_last_name \
      WHERE actor_id=::w_actor_id
   return TM_CHECK
}
if ( event == TM_INSERT_DECLARE )
{
   DBMS DECLARE :@tm_save_cursor CURSOR FOR \
      INSERT INTO actors (actor_id, first_name, last_name)\
      VALUES(::v_actor_id, ::v_first_name, ::v_last_name)\
      WHERE actor_id=::w_actor_id
   return TM_CHECK
}
return TM_PROCEED
```

The variable `@tm_save_cursor` contains the name of the cursor to be used to perform the save operations. During the handling of these events, the transaction models execute the cursor whose name is stored in `@tm_save_cursor`.

In the execution of the cursor, the bind variables (for example, `::v_first_name`) are matched to actual data by assuming that the bind variable name is the column name preceded by a prefix, as follows:

Use	**Prefix**	**Example**
`WHERE` clause	w_	`w_actor_id`
`SET` clause	s_	`s_actor_id`
`VALUES` clause	v_	`v_actor_id`

For information about how the bind variables are used in SQL generation, refer to page 33-43, "Viewing the SQL Statements."

CHAPTER

33 Using Automated SQL Generation

SQL commands are generated by the SQL generator during normal transaction manager processing. The transaction models used by the transaction manager call the SQL generator to create the appropriate SQL statements at runtime. This chapter briefly discusses the transaction manager commands corresponding to the major SQL statements.

In addition, this chapter discusses:

- How Panther uses database tables and columns to build SQL statements.
- How the SQL generator builds SQL statements using various screen and widget property values.
- Each type of SQL data manipulation statement: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, the syntax of the statement and what properties are used to define the statement construction.
- How to implement optimistic database locking.
- How to view and modify the generated SQL.
- How the transaction manager can validate data entry with the appropriate SQL generation.

For basic information about SQL and SQL construction, refer to Chapter 3, “Introduction to SQL,” in the *JDB SQL Reference*.

For information about how Panther reads data from the database, refer to Chapter 29, “Reading Information from the Database,” and for information about how Panther writes data to a database, refer to Chapter 30, “Writing Information to the Database.”

Guidelines for Automated SQL Generation

Panther uses information it gathers about your database to provide the SQL generator with the information it needs to build the appropriate SQL statements. It gathers this information via property specifications. These guidelines describe the types of information that reside in specific widgets.

Specifying Tables

Table view widgets contain most of the database table information. The following guidelines describe how Panther interprets database tables:

- The database table name used in the SQL statement is determined via the Table property (table) setting (under Database) for the table view widget associated with the database table.
- A table view corresponding to the database table must exist on the screen to ensure that the database table is included in a SQL statement.
- If there is more than one table view on a screen, links defining the relationship between table views must reside on the screen for automatic SQL generation to occur.
- The table view must have its Updatable (updatable) property under Transaction set to Yes in order to generate `SQL INSERT`, `UPDATE` and `DELETE` statements for its corresponding database table.

- Since you can apply a transaction manager command to one or more table views, all table views on a screen do not necessarily participate in each command.

Specifying Columns

The widgets in each table view generally correspond to database columns. The Database properties provide information to the SQL generator that it uses to construct the appropriate SQL statements. The following guidelines apply to database columns:

- The widget's Column Name (`column_name`) property under Database should correspond to the database column name. The column name is assigned automatically as a result of the import process.
- A correlation name (generally `tableName.columnName`) is used in the SQL statement unless database Expression subproperties are set to other values. A correlation name is used in case the column is a member of two different database tables.
- To update a column in a database table, a widget corresponding to that column must belong to the table view associated with that database table. Refer to "Manipulating Table View Members" on page 22-10 in *Using the Editors* for details on how to define a widget's membership in a table view.
- In addition to belonging to a table view, in order to participate in `SQL SELECT`, `INSERT` and `UPDATE` statements, the widget associated with the database column must have its Database subproperties set appropriately: Use In Select (`use_in_select`) under Fetch Data; Use In Insert (`use_in_insert`) under New Data, and Use In Update (`use_in_update`) under Change Data, respectively. By default, a widget that is the result of an import from the database, has Yes settings for each of these properties.

Generating SQL in the Transaction Manager

When you use the transaction manager, the SQL statements are automatically generated from the various property settings. To control or change the generated SQL, you can change the properties associated the widgets representing the database tables and their columns, or you can add event functions to handle certain transaction manager requests. For more information on writing event functions, refer to Chapter 32, “Writing Transaction Event Functions.”

The SQL generator uses Database properties to construct `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements. The transaction manager, via table view's and widget's Transaction property specifications, tells the SQL generator which of these statements to generate. Table 33-1 outlines which transaction manager commands are needed to generate the different types of SQL statements.

Table 33-1 SQL statements generated via the transaction manager

SQL statement	Transaction manager command
`DELETE`	Generated via `SAVE` command after rows are deleted or cleared of data in update mode. Table view must be updatable.
`INSERT`	Generated via `SAVE` command after new rows are inserted in update or new modes. Table view must be updatable.
`SELECT`	Generated via `VIEW` and `SELECT` commands. In order to update selected data, the `SELECT` command must be used.
`UPDATE`	Generated via `SAVE` command after data are modified in update mode. Table view must be updatable.

Many of these properties are automatically defined and set via the import process and by the screen wizard when you use it to build screens for your application. In general, Database, Transaction, and Service (JetNet and Oracle Tuxedo only) property specifications are used by the transaction manager to effect SQL generation.

The Database properties listed in Table 33-2 are associated with table views and provide the SQL generator with information it needs to generate the appropriate SQL statement for a specific table view.

Table 33-2 Table view Database properties for generated SQL

Property	Description
Table (`table`)	Name of the database table. Automatically provided if the table view was copied from a repository entry generated by the import process.
Primary Key (`primary_key`)	Columns composing database table's primary key. Generated automatically if it was avail able from the database engine.
Sort Widget (`sort_widgets`)	Sort order by which data are displayed (in ascending or descending order).
Distinct (`distinct`)	Include or omit duplicate rows from query results.

Table 33-3 lists service properties, used with the JetNet/Oracle Tuxedo middleware adapter, that specify the service to carry out each of the corresponding transaction manager commands. These Service properties are automatically set for Master table view widgets on client screens and selection screens (if any) when you use the screen wizard to generate screens.

Table 33-3 Table view Service properties for SQL statements in JetNet

Property	Description
Select Service	Name of service that implements Select operation to retrieve information stored in database table.
Insert Service	Name of service that implements Insert operation to add data to database table.
Update Service	Name of service that implements Update operation to change data stored in database table.
Delete Service	Name of service that implements Delete operation to re move rows from database table.

Sample Tables

To illustrate SQL generation, many of the examples in this chapter use the following database tables which are part of a database called vacation. The database includes three database tables: `vacations`, `customers`, and `cust_trips`.

```
CREATE TABLE vacations
(
   destination    CHAR (30)    NOT NULL,
   num_days       INTEGER,
   type_id        CHAR (10),
   travel_costs   FLOAT,
   hotel          FLOAT,
   meals          FLOAT,
   PRIMARY KEY (destination)
)

CREATE TABLE customers
(
   cust_id        INTEGER      NOT NULL,
   first_name     CHAR (20),
   last_name      CHAR (25),
   phone          CHAR (15),
   PRIMARY KEY (cust_id)
)

CREATE TABLE cust_trips
(
   cust_id        INTEGER      NOT NULL,
   destination    CHAR (30)    NOT NULL,
   paid_flag      CHAR (1),
   paid_date      DATETIME,
   PRIMARY KEY (cust_id, destination),
   FOREIGN KEY (cust_id) REFERENCES customers (cust_id),
   FOREIGN KEY (destination)
      REFERENCES vacations (destination)
)
```

Note: The SQL examples might not match the SQL generated by Panther, because the SQL might be for a specific database engine, or the sequence of items in the statements might reflect the order in which widgets are added to a screen. However, the order should not affect the results.

The examples describe which properties you need to set for each widget and table view in order to produce the necessary SQL. Most of these properties can be set in the Properties window.

Generating SELECT Statements

The SQL generator generates one `SELECT` statement per server view. A server view consists of a table view and all other table views linked to that table view with a server link. Therefore, a master-detail situation, which requires a sequential link, generates at least two `SELECT` statements, one for the master (parent) table view, and one for the detail (child) table view.

The `SELECT` statement retrieves data from a database and returns it to the user in the form of query results. The following is an example of the syntax of the `SQL SELECT` statements that can be generated by the SQL generator:

```
SELECT [ distinct-keyword ] select-list
    FROM table-list
    [WHERE where-condition]
    [GROUP BY group-by-list]
    [HAVING having-condition]
    [ORDER BY order-by-list]
```

Setting widget properties determines how each element of the SQL statement is, in fact, generated.

Table 33-4 lists the major elements of a `SQL SELECT` statement, and briefly describes what properties trigger generation of those elements. Table 33-5 lists additional SQL elements that can be generated via property specifications, event functions, or with calls to specific functions. More detailed information about `SQL SELECT` elements is presented in the sections following the tables.

Table 33-4 SELECT statement SQL element properties

SQL element	Property settings
distinct-keyword	For the table view, Distinct (distinct) property is set to Yes.
select-list	For each widget in the server view whose Use In Select (`use_in_select`) property is set to Yes. The value in the Column Name (`column_name`) property is used unless a select Expression (`select_expression`) is defined.

Table 33-4 SELECT statement SQL element properties *(Continued)*

SQL element	Property settings
`tableName`	For the table view, value set in Table (table) property.
`WHERE` clause	In applicable widgets, those having Use In Where (`use_in_where`) set to Yes and the Operator (`where_operator`) property is one of the following: =, <>, <, <=, >, >=, in, like, like%, %like%, not in, not like, not like%, not %like%
	Joins: For link widgets, type is set to `PV_LNK_SERVER`. The Relations (relations) property must contain the column names to be joined and must list join as the relation type.
`GROUP BY` clause	For aggregate functions, this clause is automatically generated. Otherwise, for applicable widgets, the column name is specified in the Group By (`group_by`) property.
`HAVING` clause	For applicable widgets, the Having (having) property specifies the search condition.
`ORDER BY` clause	For the table view, the Sort Widgets (`sort_widgets`) property specifies `the` widgets' name associated with the data base columns, followed by `ASC` or `DESC`.

Table 33-5 Additional SQL elements in SELECT statements

SQL element	Property settings
Aggregate functions	For applicable widgets, the aggregate function is specified in the select Expression (`select_expression`) property when Use In Where (`use_in_where`) is set to Yes.
`BETWEEN` predicate	Use an event function to call the function `dm_gen_change_select_where`.

Table 33-5 Additional SQL elements in SELECT statements *(Continued)*

SQL element	Property settings
`COUNT`(*) function	For the table view, Count Select (`count_select`) is set to Yes. This replaces the select clause and returns a row count before actually fetching records.
`EXISTS` clause	Use an event function to call the function `dm_gen_change_select_where`.
`IN` clause	For the applicable array, set Use In Where (`use_in_where`) to Yes and specify in (`PV_WHERE_IN`) for the Operator (`where_operator`) property. At run time, the widget should have a value before executing the `SELECT` statement.
`LIKE` predicate	For applicable widgets, Use In Where (`use_in_where`) is set to Yes and one of the like operators is specified in the Operator property (refer to the available values for the `where_operator` property). At runtime, the widget should have a value before executing the `SELECT` statement.
Null values	For applicable widgets, Use If Null (`where_use_if_null`) and Null Field (`null_field`) properties are set to Yes. Supply a value to Null Text (`null_text`).
Operators	Can only be set for `WHERE` clauses in `SELECT` statements. For applicable widgets, Use In Where (`use_in_where`) is set to Yes, and an Operator (`where_operator`) is specified.
Stored procedures	Use an event function.
Subqueries	Use an event function to call the function `dm_gen_change_select_where`.

If a desired SQL statement cannot be generated automatically, you can write a transaction event function either to supply the custom SQL or to call the SQL modification functions. For more information on writing transaction event functions, refer to Chapter 32, "Writing Transaction Event Functions."

Fetching Data from the Database

The select-list is a list of columns, expressions or aggregate functions whose values you want to fetch from the database. The select-list is derived from all of the widgets in the server view whose Use In Select property is set to Yes. Each of these widgets contributes one item to the select-list—either the value of the widget's select Expression property, if set, or the widget's Column Name.

Defining a Widget's Participation in SELECT Statements

To have the widget be a part of the *`select-list`* in a SELECT statement, set the widget's Use In Select (`use_in_select`) property, under Fetch Data, accordingly:

- Yes (default for database-derived widgets)—The database column associated with the selected widget (as specified in the Column Name property) is included in the select list of an `SQL SELECT` statement. The value in the Column Name (`column_name`) property is used unless a select Expression is defined (`select_expression`).

  If the widget's Column Name is used, it appears in the SQL statement in the following format: `tableviewName.columnName`

  Additional `SELECT`-specific subproperties are available.

- No—The database column is excluded from the select list of the `SELECT` statement.

Implementing a SELECT expression

A subproperty under the Use In Select property when Use In Select is set to Yes. Include an expression in the select list instead of the column name. The expression calculates the value to be included in the query results. The expression can be specific to a particular database engine. If you do not include an expression, the column name associated with the selected widget is used as a select item. A select expression can be, for example, an aggregate function. If you include an aggregate function, the SQL generator automatically builds a `GROUP BY` clause based on the column associated with the selected widget.

Controlling How Data Is Selected

In this example, you want the total cost of each vacation. The desired SQL is:

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
   FROM vacations
```

Figure 33-1 includes five widgets, all of which are members of a table view associated with the vacations table. Widget #5, named `total_cost`, is a derived column (that is, it is not represented in the database, but derives its data from other database columns). Widget #5 has the following property settings:

- Use In Select is set to Yes. Because a select Expression property is defined for this widget, the expression is included in the select-list of the `SELECT` statement.
- Use In Insert and Use in Update properties are set to No which prevents this derived column from being included in `INSERT` and `UPDATE` statements.

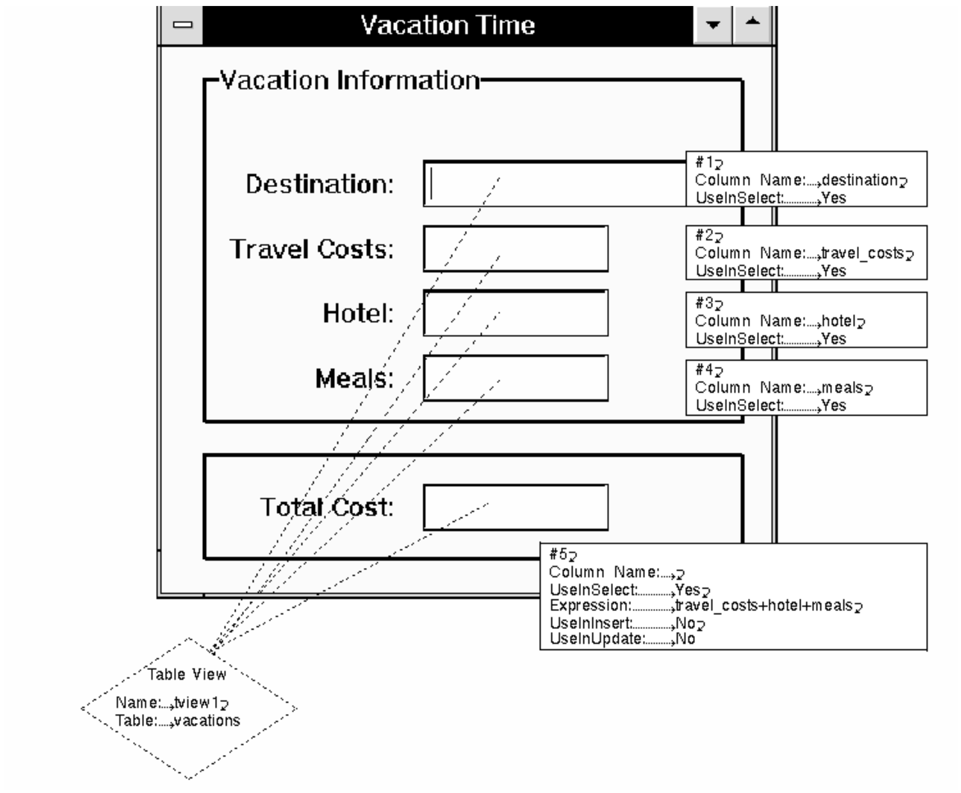


Figure 33-1 The widgets all belong to the tview1 table view, therefore, even Widget #5, which does not have a corresponding column in the vacations database table, can be included in SQL generation.

Validating Data

There are two properties you can set which will effect what happens when data are written to a widget as the result of a `SELECT` statement:

- Set Valid property—Sets the widget's content as valid so that validation is not rerun for the widget's data,
- Force Valid property—Invokes the validation function.

Setting the Widget's Contents as Valid

The Set Valid (`select_set_valid`) property is a subproperty under the Use In Select property when Use In Select is set to Yes. If the Set Valid property is set to Yes, when the `SELECT` statement fetches data to the widget, Panther sets the widget's runtime valided property. This can ensure that the content of the widget is not revalidated upon screen exit.

The valided property, when set to `PV_YES`, indicates to Panther that the widget has passed its validation tests. This occurs when the data is written to the widget, not when it finishes writing the row.

If you use the Set Valid property in conjunction with the Force Valid property, it is executed before Force Valid.

Forcing Validation of the Widget's Contents

Set the widget Force Valid (`select_force_valid`) property to Yes, a subproperty under the Use In Select property if Use In Select is set to Yes. This specification tells Panther to call the field validation function `sm_fval` after data is written to the widget. If validation displays messages, the errors are ignored and the `SELECT` statement is allowed to continue. If this property is used in conjunction with the Set Valid property, Set Valid is executed first.

Consider a client screen that fetches phone numbers from the database. You want to display those numbers in a particular format (with parentheses and hyphens, for example). By default, validation on this widget is only invoked when a user exits the field or when the screen closes. So, the format (or Keystroke Filter property specification) won't be applied to the widget's content until the user enters or exits the widget. By writing a JPL procedure that imposes a format (and specifying the procedure in the widget's Validation Function property) and setting the Force Valid property to Yes, you force the validation function, and in this case, the format, to be applied to the data when it is written to the widget.

Eliminating Duplicate Rows in a Result Set

When the select-list of a `SELECT` statement includes the primary key of a table, every row of the result set is unique (because the primary key has a different value for each row). However, if the primary key is not included in the result set, duplicate values can be returned.

How to Implement the DISTINCT or UNIQUE Keyword

Set the table view's Distinct (distinct) property, under Database, to Yes. Panther supplies the correct keyword for your database, `DISTINCT` or `UNIQUE`, and applies it to the server view, thereby eliminating duplicate values from the query results.

Determining What Tables to Select From

The table-list is a comma-separated list of all table views in the server view. The list comprises the all Table (table) property specifications. For each table, a correlation name, or alias, pairs the database table with its associated table view name.

For many databases, when a database table is imported to the repository by a user who is not the owner of the table, two table view properties, Name (under Identity) and Table (under Database), also list the owner name. In this case, the owner name appears in the table list in the format: `owner.tableName`

Defining the Where Condition

The where-condition is derived from the widgets whose Use In Where property is set to Yes. A where-condition compares data in the widget with data in the database column—specifying the rows that you want to retrieve. The Use In Where property defines how the database column associated with a selected widget is treated when the `SQL SELECT` statement is constructed and whether it is included in the statement's `WHERE` clause.

In addition, when a widget's select Expression (`select_expression`) property contains an expression and its Column Name property is blank, then the SQL generator uses the expression in the where-condition. This enables the where-condition to contain comparisons involving computed columns.

How to Define a Widget's Participation in the WHERE Clause

To have a widget participate in the WHERE clause of a SELECT statement:

1. Set the widget's Use In Where (`use_in_where`) property (under Fetch Data) accordingly:

- Yes—The data in the widget is compared to the data in its associated database column. If more than one widget on the screen has this setting, the AND keyword is used to join the conditions to build the `WHERE` clause. Related subproperties are displayed.
- No (default)—The widget's data is not included in the `WHERE` clause of the `SELECT` statement.

2. Set the Operator (`where_operator`) subproperty to the desired operator to use in the `WHERE` condition. The default operator is = (equal to). The supported operators are:
   - For comparison tests, = (equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to), or <> (not equal). Comparison tests compute and compare values of two SQL expressions for each row of data.
   - For set membership tests, not or not in. The `WHERE` clause tests for a data value that matches the target value.
   - For pattern matching, like, like%,%like%, not like, not like%, or not%like%. The percent sign (%) wildcard character matches any sequence of zero or more characters. Use the "not like" operators to locate strings that do not match a pattern.

   Note: Pattern matching can only be performed if the widget's C Type property specification is character string (`PV_CHAR_STRING`).

3. Set the Use If Null (`where_use_if_null`) subproperty to desired setting to define how null values should be treated:
   - Yes—If the widget has a null value, it is included in the `WHERE` condition. In which case, blank data are treated as a null database value. You can change the widget's representation of null data by changing the widget's Null Field (`null_field`) property under Format/Display.
   - No—(default) If the widget has blank or null (where null is defined by the widget's null edit) data, the widget does not contribute to the `WHERE` condition.

he column name is derived from the widget's Column Name (`column_name`) property. The comparison operator is the value specified in the Operator (`where_operator`) property when Use In Where is set to Yes.

If the widget is an array, the value used for the operator must be entered in the first occurrence of the array, except for the in operator. The in operator uses the data in all of the array occurrences to construct the `IN` clause.

Fetching an Exact Match

For example, to get the total cost for a particular destination, the desired SQL is:

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
   FROM vacations
   WHERE destination = destination
```

Using Figure 33-1 on page 33-12, Widget #1 (associated with the column destination) has its Use In Where property set to Yes and the Operator set to =. This allows a user of the application to enter a desired destination, execute a `SELECT` or `VIEW` command, and as a result fetch only those rows from the vacations table where the destination column data matches (or equals) the value entered in Widget #1, destination.

Fetching Records Matching a Partial String

If you specify the Operator property to be `like%`, you can use the pattern matching capability of the database to search for the desired destination. Changing the example for using the = operator to like results in:

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
   FROM vacations
   WHERE destination LIKE destination%
```

This allows the user to enter a partial string in Widget #1, for example Lon. When the `SELECT` or `VIEW` command is executed, all destinations beginning with those letters are fetched from the database.

Fetching Records Matching One of a List of Values

If you specify `in` for the where Operator, and change the single line text destination widget to an array greater than one, a series of destinations can be entered for database searches. Your query tests whether the database values match one of the listed values in the `WHERE` clause.

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
```

```
    FROM vacations
    WHERE destination IN (destination, destination, ...)
```

The user can enter a destination in each occurrence before executing the `SELECT` or `VIEW` command, and the fetched data will be for all specified destinations.

Fetching Null Values

Normally, widgets whose Null Field property is set Yes and whose data are blank or null do not contribute to the where-condition. To force widgets containing null data to contribute, set the Use If Null (`where_use_if_null`) property to Yes. Blank, or empty data are treated as null database values and a `WHERE` clause is generated.

The text of the `WHERE` clause depends on the setting for the Operator property. To test for a null value, the operator should be set to = (equal to), then the query checks for `NULL` values and builds that `WHERE` clause as: `WHERE` column `IS NULL`. To query for those rows that do no contain null values, set the operator to <> (not equal to), the SQL generator builds the `WHERE` clause as: `WHERE` column `IS NOT NULL`.

Selecting Null Data

List those rows where a hotel cost has not be determined. The desired SQL is:

```
SELECT destination, travel_costs, hotel, meals,
    travel_costs+hotel+meals
    FROM vacations
    WHERE hotel IS NULL
```

To ensure that all rows are returned in the result set, including those where hotel costs are not specified, the widget corresponding to the database column hotel should have its Use In Where property set to Yes, the Operator property set to =, and the Use If Null property set to Yes.

Grouping SELECT Statement Results

The SQL generator automatically builds a `GROUP BY` clause if any widget's select Expression (`select_expression`) property uses one of the aggregate functions: `AVG`, `COUNT`, `SUM`, `MIN`, `MAX` (no other aggregate functions are automatically detected). When an aggregate function is detected, the group-by-list automatically includes the column name of every widget in the server view whose Use In Select (`use_in_select`) property is set to Yes and select Expression does not implement an aggregate function.

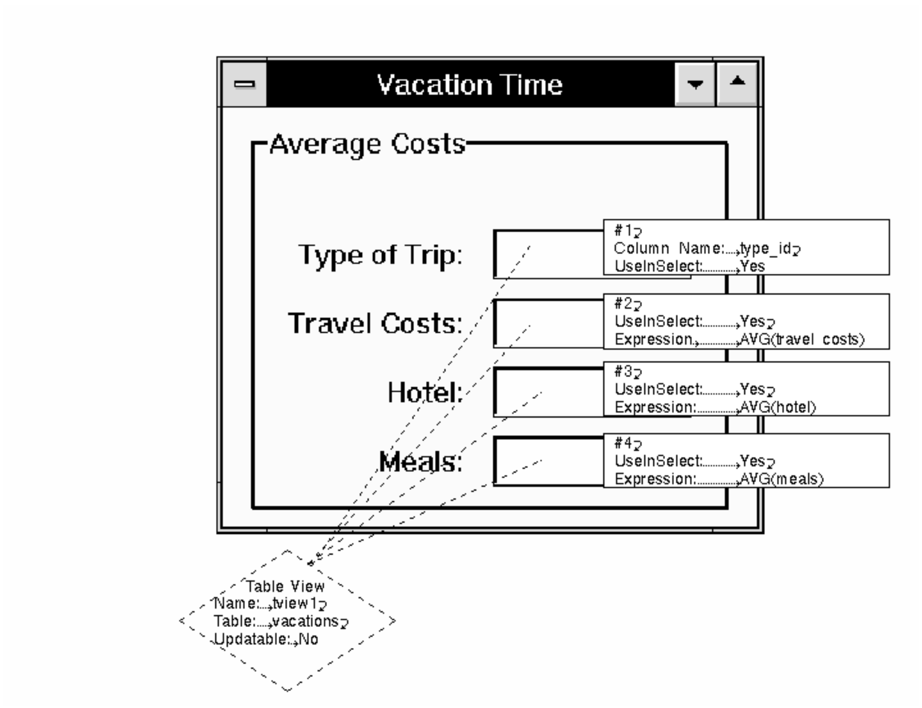
Warning: If any one widget contains an automatically detected aggregate function, and a second widget on the screen contains an undetected aggregate function, then Panther adds the second widget's column name to the group-by-list. To prevent this from happening, clear that widget's Column Name property.

Grouping Results Automatically

Get the average travel, hotel and meal costs, grouped by type of trip. The desired SQL is:

```
SELECT type_id, AVG(travel_costs), AVG(hotel), AVG(meals)
   FROM vacations
   GROUP BY type_id
```

Figure 33-2 includes four widgets, all are members of a table view associated with the table vacations. The table view's Updatable (updatable) property is set to No to prevent update and insert attempts to that database table. Widgets 2, 3, and 4 all implement an aggregate function (`AVG`), therefore, they are included in the `select_list` and the remaining column is automatically included in the `GROUP BY` clause.



#1
Column Name: type_id
UseInSelect: Yes

#2
UseInSelect: Yes
Expression: AVG(travel_costs)

#3
UseInSelect: Yes
Expression: AVG(hotel)

#4
UseInSelect: Yes
Expression: AVG(meals)

Table View
Name: view1
Table: vacations
Updatable: No

Figure 33-2 The `SELECT` statement groups the results by `type_id`, since the other three widgets specify a select Expression that use the `AVG` aggregate function.

Specifying a GROUP BY Clause

When the SQL generator cannot detect the presence of an aggregate function in one of the widgets' select Expression property, you must set a widget's Group By (`group_by`) property, under Fetch Data. Enter the names of the columns whose values are to be used to group the data.

For example, get the standard deviation of the total cost, grouped by type of trip. The desired SQL is:

```
SELECT type_id, STDDEV(travel_costs+hotels+meals)
    FROM vacations
    GROUP BY type_id
```

Consider an application screen that contains two widgets, both are members of a table view associated with the vacations database table. The table view's Updatable (updatable) property is set to No to prevent updates and inserts to the database table. One widget on the screen, named standard, simply displays the standard deviation of the total cost of a trip. It does this by having a select Expression setting of STDDEV(travel_costs+hotel+meals). The widget associated with the type_id column has its Use In Select property set to Yes, and must also have its Group By (group_by) property explicitly set to type_id, because widget standard contains an aggregate function that is not automatically detected by the SQL generator.

Grouping Multiple Columns

To group query results based on the contents of two or more columns, specify multiple column names in the Group By property. You can also designate columns not included in the SELECT statement's select-list.

For example, get the average total cost of each destination, grouped by their travel costs and their type. The desired SQL is:

```
SELECT travel_costs, AVG(travel_costs+hotel+meals)
   FROM vacations
   GROUP BY travel_costs, type_id
```

Consider an application screen that contains two widgets: travel_costs and travel_total; both are members of the table view associated with the vacations database table. The travel_costs widget has its Use In Select property set to Yes; it will be included in the group-by-list of the SELECT statement. The travel_total widget also has its Use In Select property set to Yes and a select Expression defined as AVG (travel_costs+hotel+meals). Even though Panther detects the presence of an aggregate function (AVG), the Group By property on this widget needs to be set to type_id, because none of the widgets in the table view correspond to the type_id column. The column associated with the travel_total widget will not be included in the group-by-list, but its group-by specification is.

Applying Search Conditions to the Result Set

The having-condition of the SELECT statement applies an additional search condition once the result rows have been determined. Generally, the HAVING clause appears in conjunction with a GROUP BY clause.

The having-condition is derived from the widgets in the server view whose Having (having) property is set. If more than one widget in the server view has this setting, the `AND` keyword is used to join these conditions.

Specifying an Aggregate Function in the HAVING Condition

Get the average vacation cost, grouped by type and only report those types whose average cost is below 1000. The desired SQL is:

```
SELECT type_id, AVG(travel_costs+hotel+meals)
   FROM vacations
   GROUP BY num_days
   HAVING AVG(travel_costs+hotel+meals) < 1000
```

The widget that will display the total results has its Having property, under Fetch Data, set to `AVG`(`travel_costs`+hotel+meals) < 1000.

Sorting the Results from a SELECT Statement

The order-by-list sorts the result rows according to the values in specified columns. The order-by-list is built from the table view's Sort Widgets (`sort_widgets`) property.

Specifying a Sort Order for a Specific Table View

Under Database, in the Sort Widgets property, enter a list of widget names and an optional order specifier (case-insensitive), one per line. The order specifier should be separated from the widget name by a space. Valid order specifiers are:

`DESC`—Descending order
`ASC`—Ascending order

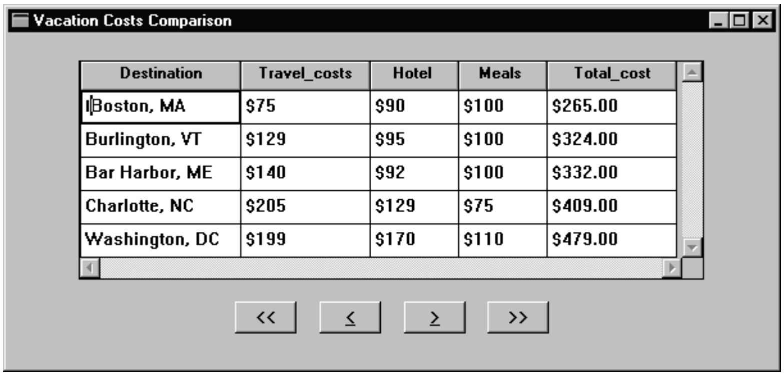
When the SQL is generated, Panther specifies the sorting order in a manner acceptable to the database engine. If no order specifier is entered, the results display in ascending order. The SQL generator uses the widget name to determine the associated column or select expression to be sorted.

Example: Sorting Results

Get the total cost of each vacation and order the costs in ascending order. The desired SQL is:

```
SELECT destination, travel_costs, hotel, meals, total_cost
    FROM vacations
    ORDER BY total_cost ASC
```

The screen illustrated in Figure 33-3 includes widgets in a grid that represent four database columns: destination, `travel_costs`, hotel, and meals. The screen also includes a widget (`total_cost`) that derives its data from its select Expression (`travel_costs`+hotel+meals). All five widgets have their Use In Select property set to Yes and are members of the vacations table view which has its Sort Widgets property set to `total_cost` desc. The result set lists all vacations, by cost, in ascending order—the least expensive listed first.



Destination	Travel_costs	Hotel	Meals	Total_cost
Boston, MA	$75	$90	$100	$265.00
Burlington, VT	$129	$95	$100	$324.00
Bar Harbor, ME	$140	$92	$100	$332.00
Charlotte, NC	$205	$129	$75	$409.00
Washington, DC	$199	$170	$110	$479.00

Figure 33-3 Results to the query can display the total_cost in descending order.

Selecting Data from Multiple Database Tables

In general, joins are built by comparing pairs of columns from two joined tables by testing the data from both columns for equality or other comparisons. Sometimes these joins have a one-to-one relationship while others have a one-to-many relationship. Most common multi-table queries use parent/child relationships created by primary keys and foreign keys. Via the link widget's properties, Panther lets you define the join relationship and the join type between table views. The Relations property specifies the columns or widgets that connect two table views and the Join Type property (for server links) lets you define the type of join: inner, right or left outer, or full outer.

The number of `SELECT` statements issued by the SQL generator depends on the type of link specified in the link widget properties. For instance, table views that are joined by server links cause Panther to issue a single statement with a join in the `WHERE` clause of the `SELECT` statement. For table views that are connected by sequential links, Panther issues multiple statements using values fetched in the parent table view to create the where-condition in the child table view.

The database column or columns needed to construct a SQL join for the two table views are defined by three link widget properties: Type, Join Type (a subproperty of the Type property, and Relations property.

How to Specify the Join Relationship

1. Under Transaction, specify the link widget's Type property as one of the following:

   - Sequential—To join tables that have a one-to-many relationship.
   - Server—To join tables that have a one-to-one relationship, or to display multiple records using a single condition.

     The Join Type property is displayed where you can specify inner or outer join types (refer to “Specifying the Join Type” on page 33-28 for details on specifying join types).

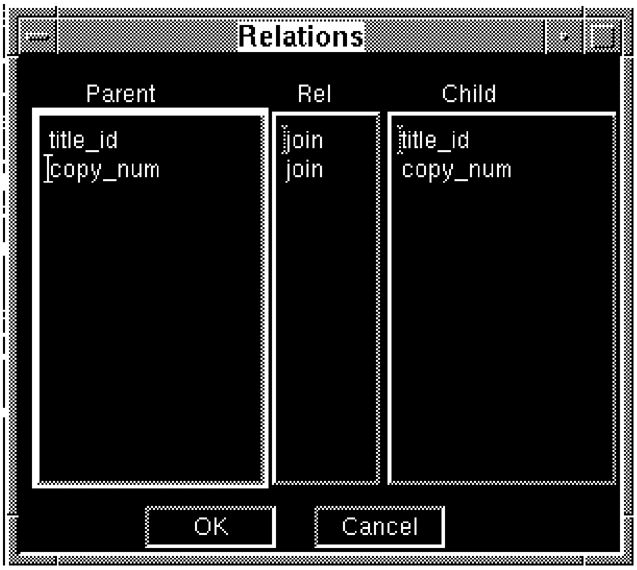
2. Select the Relations property to define the join relationship. In the Relations dialog box, specify the Parent (`rel_parent`) and Child (`rel_child`) column names (column names are case-sensitive). The column specifications are used to build the SQL join condition in the `WHERE` clause.

   Choose Help to display and select from a list of columns associated with parent and/or child database tables.

   For sequential links, if the column specified in the Parent list of the Relations property is represented by more than one widget in the table view or on the screen, the widget's name is used instead of the column name; using the following format (including the square brackets and the literal +0):

   ```
   ::widgetName[+0]
   ```

This ensures that the value returned to the named widget (belonging to the parent table view) is used in the SQL statement for the child table view.



Parent	Rel	Child
title_id	join	title_id
copy_num	join	copy_num

Figure 33-4 The Relations dialog box lists the parent and child relationships for the selected link.

3. Define the type of relationship—join—in the Rel (`rel_op`) list of the Relations dialog box. The only relationship that can be specified for joins is the word join.

The `WHERE` clause of the statement will include for each join relation specified, one expression of the form:

- For sequential links: `widgetData_in_parentTableview = childTable.childColumn`
- For server links: `parentTable.parentColumn = childTable.childColumn`

If there are multiple joined columns, then the expressions are connected by the keyword `AND`. If more than one table view in the server view represents the same database table, the SQL generator automatically supplies table alias names as needed. Therefore, self-join expressions are automatically handled.

Specifying Joins in the Where Condition

For equi-joins, joins where the operator is = (which includes self-joins), the link between the joined table views specifies Server as the type of link and Inner has the join type. The Relations property contains the names of the columns included in the `WHERE` clause. Specify the relationship between the parent and child columns as join. For each join relation specified, the where condition includes one expression of the `formparentTable.parentColumn = childTable.childColumn`

Implementing an Equi-join: one-to-one relationship

Join each customer's name and trip destination. The desired SQL is:

```
SELECT customers.cust_id, first_name, last_name, destination
   FROM customers, cust_trips
   WHERE customers.cust_id = cust_trips.cust_id
```

In Figure 33-5, the screen contains three widgets, all of which are members of the `tview1` table view which is associated with the customers database table: `cust_id`, `first_name`, and `last_name`. A fourth widget, destination, belongs to the `tview2` table view which is associated with the `cust_trips` database table.

The link (tview1+tview2) between the two table views has a Type property setting of Server. The Relations property sets `cust_id` in the parent table view is to be joined with `cust_id` in the child table view. The result of these specifications causes a single `SELECT` statement to be generated which populates both the parent and child table views. Therefore, for every customer ID, a destination is displayed.

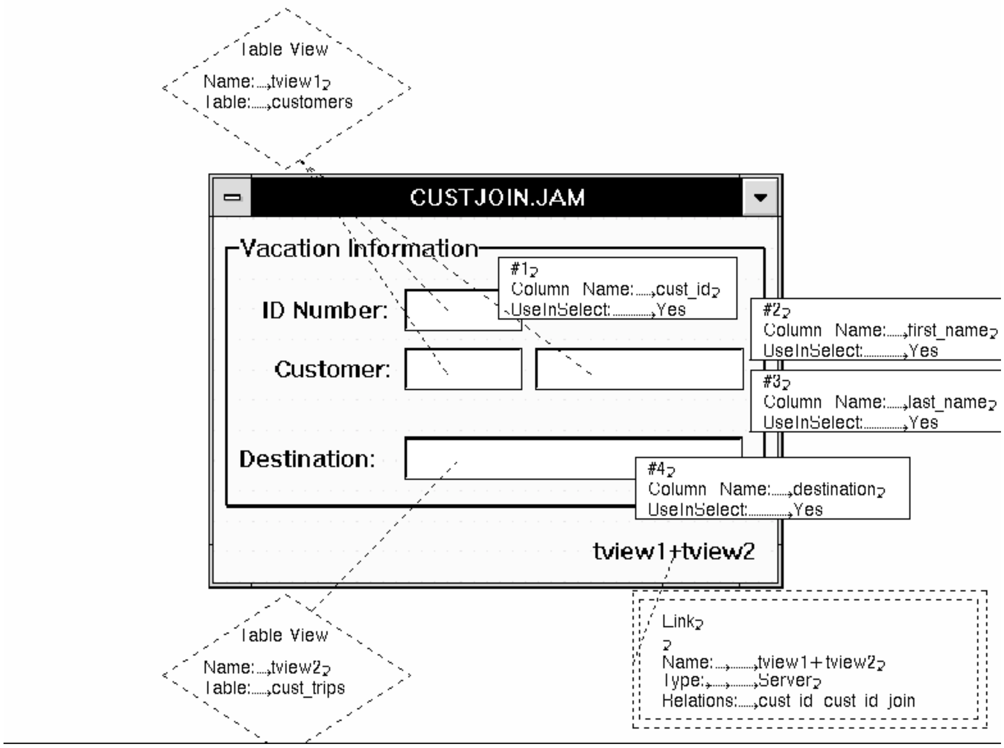


Table View
Name:....tview1
Table:......customers
CUSTJOIN.JAM
Vacation Information
ID Number:
Customer:
Destination:
#1
Column Name:......cust_id
UseInSelect:.............Yes
#2
Column Name:......first_name
UseInSelect:.............Yes
#3
Column Name:......last_name
UseInSelect:.............Yes
#4
Column Name:......destination
UseInSelect:.............Yes
tview1+tview2
Table View
Name:....tview2
Table:......cust_trips
Link
Name:..........tview1+tview2
Type:..........Server
Relations:......cust id cust id join

Figure 33-5 A server link defines a one-to-one relationship between two table views.

Generating Multiple SELECT Statements: One-to-many Relationship

List each customer's trip destinations. The desired SQL is:

```
SELECT cust_id, first_name, last_name, phone FROM customers

SELECT destination FROM cust_trips

    WHERE cust_trip.cust_id = value in customers.cust_id
```

When the link's Type property is specified as Sequential, Panther generates one SQL `SELECT` statement for the parent table view, and one for the child. Sequential links must be specified for master-detail screens where there are several detail rows associated with one master row. For sequential links, the SQL for the child's where-condition contains an expression similar to:`widgetData-in-parentTableview = childTable.childColumn`

The link's Relations property must specify both a column in the child table view, and a widget or column in the parent table view.

In Figure 33-6, the screen contains three widgets, all of which are members of the `tview1` table view which is associated with the `customers` database table: `cust_id`, `first_name`, and `last_name`. A fourth widget, `destination`, is grid widget that displays more than one occurrence, and belongs to the `tview2` table view which is associated with the `cust_trips` database table.

The link (tview1+tview2) between the two table views has a Type property setting of Sequential. The Relations property sets `cust_id` in the parent table view is to be joined with `cust_id` in the child table view. The result of these specifications causes a single `SELECT` statement to be generated for the parent table view which populates the parent table view, and given that result, uses the value returned to generate the next `SELECT` statement to populate the child table view. Therefore, for every customer ID, all destinations associated with that customer ID are displayed.

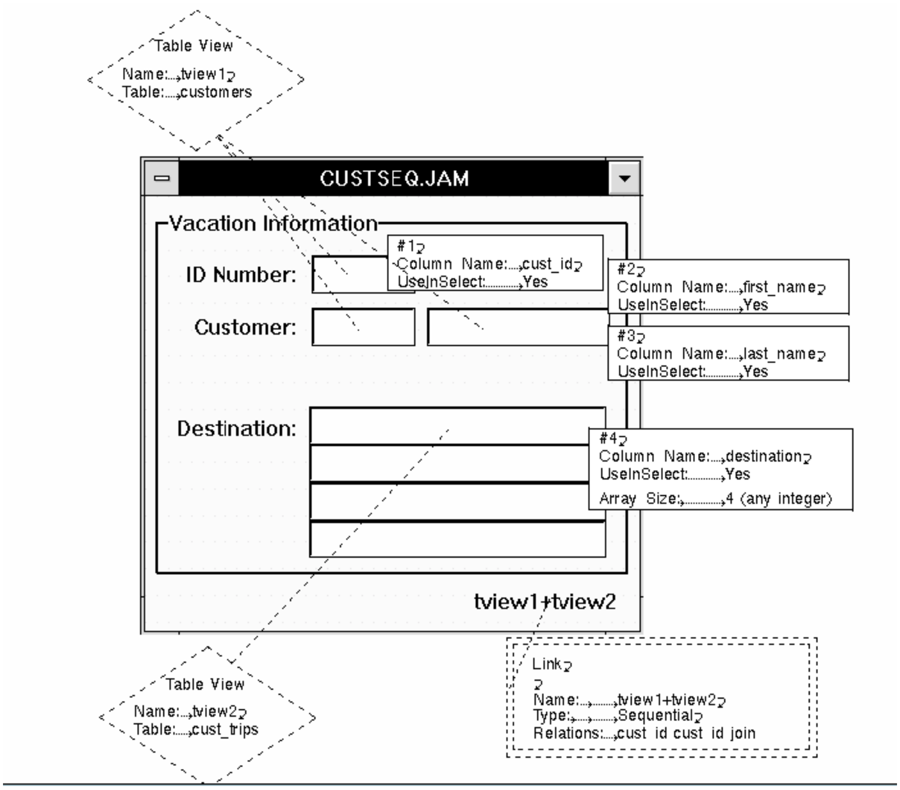


Table View
Name:....tview1
Table:.....customers
CUSTSEQ.JAM
Vacation Information
ID Number:
Customer:
Destination:
#1
Column Name:....cust_id
UseInSelect:..........Yes
#2
Column Name:....first_name
UseInSelect:..........Yes
#3
Column Name:....last_name
UseInSelect:..........Yes
#4
Column Name:....destination
UseInSelect:..........Yes
Array Size:............4 (any integer)
tview1+tview2
Table View
Name:...tview2
Table:.....cust_trips
Link
Name:.........tview1+tview2
Type:..........Sequential
Relations:....cust id cust id join

Figure 33-6 A sequential link defines a one-to-many relationship between two table views.

Specifying the Join Type

A join operation combines information from two database tables by forming pairs of related rows. Since the matching columns in each table must be specified in the where-condition, you can only specify a join type (that is, a Join Type property value) if the link widget is defined as a Server type.

The link's Join Type (`join_type`) property lets you define: inner, left outer, right outer, and full outer joins. Each type returns some or all rows that meet the where-condition of the `SELECT` statement. These results can be particularly useful for calculations and aggregate functions.

Notes: Support of outer joins is dependent, and varies, on each database engine. In addition, some databases define outer joins in the `FROM` clause and some in the `WHERE` clause. If the database engine does not support the specified join type, Panther's database driver returns an error.

To illustrate the differences between the join types, consider two database tables, T1 and T2; each have two columns (Name and State):

Table T1:	Name	State	Table T2:	Name	State
	Alice	NY		Joe	MA
	Joan	ME		Fred	NY
	Paula	MA		Paul	CT
	Lynn	NY		Mike	NH

A SELECT statement that joins the two tables where T1.State=T2.State would produce different results depending on the join type.

Implementing an inner join

An inner join (default) compares two tables and fetches all possible pairs, but excludes those rows that do not meet the matching column condition for the join. For example, in Figure 33-5 on page 33-26, the results will include only those customers who have actually specified a destination; that is, the results do not include those customers where a corresponding `cust_trip` record does not exist.

In the example of joining tables T1 and T2, the results would be:

Alice NY	Fred NY
Paula MA	Joe MA
Lynn NY	Fred NY

Implementing a full outer join

A full outer join treats both database tables in the where-condition equally. While an inner join would result in only those rows that have a match, an outer join includes rows from both tables even if when there is no (NULL) match.

Given tables T1 and T2 where the SELECT statement specifies a full outer join, the results include all rows from both tables and substitutes NULL values for those rows that do not have a match:

Alice NY	Fred NY
Joan ME	NULL NULL
Paula MA	Joe MA
NULL NULL	Paul CT
Lynn NY	Fred NY
NULL NULL	Mike NH

Implementing a left outer join

A left outer join specification makes the left database table (the table referenced on the left of the equal sign in the where-condition) dominant. It compares the two tables, fetches all possible pairs and also those rows from the left table having no matching value in the right database table (`NULL` values are used). If the link's Join Type is set to Left Outer, the screen illustrated in Figure 33-5 on page 33-26 will display all customer records, even those that have not named a destination. In other words, there is no matching customer ID in the `cust_trips` database table, and therefore the destination field will be blank for some customers.

In the example of joining tables T1 and T2 where the `SELECT` statement specifies a left outer join, the results include all rows from the T1 table that have a match in T2 as well as those rows that do not—substituting `NULL` values for those rows that do not have a match in table T2:

Alice NY	Fred NY
Joan ME	`NULL NULL`

Paula MA	Joe MA
Lynn NY	Fred NY

Implementing a right outer join

A right outer join specification makes the right database table (the one on the right of the equal sign in the where-condition) dominant by fetching all possible pairs and including unmatched rows from the rightmost database table using `NULL` values if no match is found in the left table.

In the example of joining tables T1 and T2 where the `SELECT` statement specifies a right outer join, the results include all rows from the T2 table that have a match in T1 as well as those rows that do not—substituting `NULL` values for those rows that do not have a match in table T1:

Alice NY	Fred NY
Paula MA	Joe MA
`NULL NULL`	Paul CT
Lynn NY	Fred NY
`NULL NULL`	Mike NH

Modifying SELECT Statements

Automatically generated SQL statements might require additional modifications that cannot be set with widget, table view, or link properties. You can write a transaction event function to provide the desired SQL.

For SQL SELECT statements, you can also use the C functions Panther provides, listed in Table 33-6.

Table 33-6 C functions that modify generated SELECT statements

Function	Description
`dm_gen_change_execute_using`	Add or replace a bind value in `DBMS EXECUTE` statement.
`dm_gen_change_select_from`	Edit `FROM` clause in `SELECT` statement.
`dm_gen_change_select_group_by`	Edit `GROUP BY` clause in `SELECT` statement.
`dm_gen_change_select_having`	Edit `HAVING` clause in `SELECT` statement.
`dm_gen_change_select_list`	Edit select list in `SELECT` statement.
`dm_gen_change_select_order_by`	Edit `ORDER BY` clause in `SELECT` statement.
`dm_gen_change_select_suffix`	Append text to the end of `SELECT` statement.
`dm_gen_change_select_where`	Edit `WHERE` clause in `SELECT` statement.

For more information on each function, refer to the *Programming Guide*. For more information on writing transaction event functions, refer to Chapter 32, “Writing Transaction Event Functions.”

Generating INSERT Statements

An `INSERT` statement enters a new row into a database table. The SQL generator executes an `INSERT` statement for a single table, and only for table views that are updatable (updatable property is set to Yes).

If a screen contains more than one table view, the link's Insert Order (`insert_order`) property determines whether the statement for the parent table view or the child table view is generated first.

The `INSERT` statement can be generated; the SQL elements are specified in various widget and table view properties. If a certain SQL element is not supported, you can write a statement to utilize that element as part of a transaction event function.

```
INSERT INTO tableName [ (column-list) ] VALUES (value-list)
```

Table 33-7 INSERT statement SQL element properties

SQL Element	Property Settings
`tableName`	For the table view, value of the Table (table) property.
`column-list`	For each widget in the table view whose Use In Insert property is set to Yes, the value in the Column Name (`column_name`) property.
`value-list`	Contains a value for each column in the column list taken from the current widget data. If an insert Expression property specification is provided, it is used in place of the data in the widget. If the data in a widget is null, then Panther supplies an appropriate representation of null for the database.
Subqueries	Use transaction event function.

Inserting Data to Specific Columns

The column-list determines which database columns will have data entered into the database. To be included in the column-list, the widget's Use In Insert property must be set to Yes and its database column listed in the Column Name property (it cannot be blank).

Defining a Widget's Participation in an INSERT Statement

Set the widget's Use In Insert (`use_in_where`) property (under New Data):

- Yes (default)—The widget is included in the `INSERT` statement and, therefore, its datum is added to the database table with which it is associated. An Expression subproperty is displayed.

  If the selected widget is in a non-updatable table view, the Use in Insert specification is ignored.

- No—The widget is not included in the INSERT statement. Its datum is not add to the database.

Inserting Specific Values

If a column is included in the column-list, a value is entered for that column in the value-list. The value is taken from the current widget data unless an insert Expression property is defined.

Expression (insert_expression) Property

A subproperty under the Use in Insert property when Use In Insert is set to Yes. You can define an expression that is included in the value list of an INSERT statement. If you do not include an expression, the widget's data is used in the value list of the generated SQL statement.

An insert expression can be used, for example, to insert the current time, as determined by the database server. The expression can be any SQL expression that is valid for the database engine and for the database column into which data is being inserted.

Inserting Data Using an INSERT Expression

Figure 33-7 illustrates how data values can be inserted into the customers and cust_trips tables. To provide an example of an insert expression, the paid_flag widget's data will always be entered as Y. The desired SQL is:

```
INSERT INTO customers (cust_id, first_name, last_name, phone)
   VALUES (cust_id, first_name, last_name, phone)

INSERT INTO cust_trips

   (cust_id, destination, paid_flag, date_paid)
   VALUES (cust_id, destination, 'Y', date_paid)
```

The screen in Figure 33-7 includes four widgets in the master section all belonging to the table view associated with the customers table: cust_id, first_name, last_name, and phone. Since data is to be inserted into the database table, the table view (tview1) must have its Updatable property set to Yes, and the widgets corresponding to the table's primary keys (cust_id) must be on screen.

The detail section of the screen includes three widgets who are members of a table view (`tview2`) associated with the `cust_trips` table. Since data is to be inserted into the database table, the table view must be updatable and the widgets corresponding to the primary keys (`cust_id` and destination) must be on screen. All widget's in the detail section have their Use In Insert property set to Yes. The data entered at runtime will be inserted into the database. However, the `paid_flag` widget Expression property specifies that its value is “Y.”

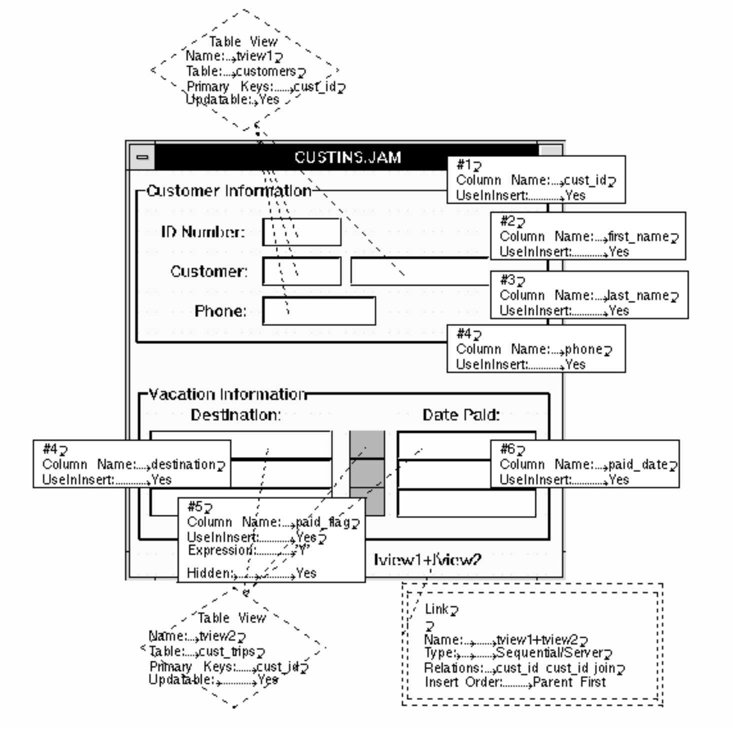


Table View
Name:...tview1
Table:...customers
Primary Keys:......cust_id
Updatable:.Yes

#1
Column Name:...cust_id
UseInInsert:...........Yes

#2
Column Name:...first_name
UseInInsert:...........Yes

#3
Column Name:...last_name
UseInInsert:...........Yes

#4
Column Name:...phone
UseInInsert:...........Yes

#4
Column Name:....destination
UseInInsert:...........Yes

#5
Column Name:...paid_flag
UseInInsert:...........Yes
Expression:..........'Y'
Hidden:...............Yes

#6
Column Name:...paid_date
UseInInsert:...........Yes

Table View
Name:...tview2
Table:...cust_trips
Primary Keys:......cust_id
Updatable:............Yes

Link
Name:.........tview1+tview2
Type:..........Sequential/Server
Relations:...cust_id cust_id join
Insert Order:.........Parent First

Figure 33-7 Property settings used to insert data into the database

Generating UPDATE Statements

An UPDATE statement updates column values in a database table. The standard models in the transaction manager generate UPDATE statements for each updatable table view if widget data in that table view has been changed.

If a screen contains more than one table view, the link's Update Order property determines whether the statement for the parent table view or the child table view is generated first.

The UPDATE statement can be automatically generated; the SQL elements are specified via various widget and table view properties. If a certain SQL element is not supported, you can write a statement to utilize that element as part of a transaction event function.

```
UPDATE tableName SET columnName = value [ , ... ]
   WHERE primary-key = before-image-data
```

The UPDATE statement modifies existing data in the database; a SET clause specifies the column or columns to update and an expression. The columns listed in the SET clause of the statement are derived from all of the widgets in the table view whose Use In Update property, under Change Data, is set to Yes. The SQL generator uses the named table view's primary key to build the WHERE clause.

Table 33-8 UPDATE statement SQL element properties

SQL Element	Property Settings
`tableName`	For the table view, value of the Table (table) property.
`columnName`	For each widget in the table view, the value in the Column Name (`column_name`) property. Use In Update property must be set to Yes.
`value`	Current widget data. If an update Expression is provided, then it is used in place of the data in the widget. If the data in a widget is null, Panther supplies an appropriate representation of null for the database.

Table 33-8 UPDATE statement SQL element properties

SQL Element	Property Settings
`WHERE clause`	Columns specified in the table view's Primary Key property.
`before-image-data`	Data in the widgets which correspond to the primary key of the table before changes were made. This might not be the values currently stored in the widget.

Identifying Columns to Update

The `SET` clause of an `UPDATE` statement specifies the column or columns to update.

Defining a Widget's Participation in an UPDATE Statement

Set the widget's Use In Update (`use_in_update`) property:

- Yes (default for database-derived widgets)—The column associated with the selected widget is included in the `SET` clause of an `UPDATE` statement. If the selected widget is in a non-updatable table view, the Use in Update specification is ignored.
- No—The column associated with the selected widget is not included in the `UPDATE` statement and, therefore, the column's datum is not changed in the database.

The `columnName` in the SQL statement is derived from the widget's Column Name. The new-value is the value currently in the widget, unless an update Expression property is set. If the Expression is set, it overrides the value in the widget.

Expression (update_expression)

A subproperty under the Use In Update property when Use In Update is set to Yes. Define an expression that is used in the `SET` clause of the `UPDATE` statement. An update expression can be any SQL expression that is valid for the database engine and for the database column being updated.

Specifying the Record to Update

The primary-key is derived from the table view's Primary Key (`primary_key`) property. The primary key (or combination) listed in the property is included in the `WHERE` clause of the `UPDATE` statement.

For example, to update the phone number for a given customer, the desired SQL is:

```
UPDATE customers SET phone = new_phone,
   WHERE cust_id = cust_id
```

When updating records, the screen must contain the widgets that represent each member of the primary key specification. In this example, the screen needs a minimum of two data widgets: one for the customer ID (`cust_id`), the primary key, and one for the customer's phone number. It also needs the table view widget associated with the customers database table.

Generating DELETE Statements

A `DELETE` statement removes rows from a database table. The SQL generator executes a `DELETE` statement only for updatable table views.

If a screen contains more than one table view, the link's Delete Order property determines whether the statement for the parent or child table view is generated first.

The `DELETE` statement can be automatically generated; the SQL elements are specified via various widget and table view properties. If a certain SQL element is not supported, you can write a statement to utilize that element as part of a transaction event function.

```
DELETE FROM tableName WHERE primary-key = before-image-data
```

Table 33-9 DELETE statement SQL element properties

SQL element	Property settings
`tableName`	For the table view, value of the Table (table) property.

Table 33-9 DELETE statement SQL element properties

SQL element	Property settings
`WHERE` clause	The columns specified in the table view's Primary Key (`primary_key`) property.
`before-image-data`	Data in the widgets corresponding to the primary key of the table before changes were made. These might not be the values currently displayed in the widget.

Refer to the element details for generating `UPDATE` statements.

Implementing Optimistic Locking

Applications, which access a multi-user database, use locking to solve concurrence problems, ensure data integrity, and data consistency. A lock is a mechanism that prevents destructive interaction between users accessing the same data. For example, two users of the application select the same customer record. The first user modifies some data and then saves the changes. The second user, unaware that the customer record has changed, makes other changes and saves those edits. Without proper locking, the second user's changes overwrite the first user's changes. To solve this problem, applications typically use one of the following locking styles:

- Pessimistic locking—The application requires an exclusive lock on data. Typically, a DBMS-specific SQL statement is executed to establish the lock before the select is executed. This prevents the second user from reading or changing the data until the application ends its transaction.

- Optimistic locking—The application does not set any database locks on the data, but checks that the selected record has not changed before it updates the record. This ensures that changes are not overwritten.

Using an exclusive lock prevents concurrent transactions from overwriting a user's changes, but it also prevents read-only transactions from viewing data. This can cause performance degradation if many users are trying to access and use the same data. An optimistic lock requires some additional setup, but it improves access and performance.

There are several ways to implement optimistic locking. Typically, as a database designer, you assign a special column to each table that will use optimistic locking. The column maintains a version number which is updated each time a row is changed. A version number is usually an integer, float, or character string, and is supported by any database. Other database-specific data types, such as time-stamping, can be useful for optimistic locking.

Refer to the *Database Drivers* for your database's information on data types.

The transaction manager provides automatic support for numeric version columns. When you set the Version Column property on a widget, Panther ensures when any of the following transaction manager statements are executed, optimistic locking is implemented as described:

- An `INSERT` statement for the widget's table view initializes the version column to 1.
- An `UPDATE` statement for the widget's table view increments the version number and includes the version number in the `WHERE` clause. Therefore, if the select returns `cust_id`=100 and `version_id`=1, and then the customer record is updated with a new address, the `UPDATE` statement would be: update customer set `address2`="Suite 200", `version_id`=2 where `cust_id`=100 and `version_id`=1;

  If another user has just updated the same customer record, the `version_id` would have incremented to 2 for that update transaction. So, the preceding `UPDATE` statement would fail, because the `version_id` 1 no longer exists.
- A `DELETE` statement for the widget's table view includes the version number in the `WHERE` clause.

Implementing Optimistic Locking using the Version Column Property

When you import database columns from the database, the version column is also imported and is represented as a widget, just like any other widget, on your screens. Make the following changes to those widgets in repository entries that represent the database version column:

1. Under Identity, ensure that the C Type property is sent to either Int, Long Int, Float, Double, Char String.

   If the column is of another type, refer to the *Database Drivers* for your database's information on data types.

2. Under Database, set the Version Column (`version_column`) property to Yes.

3. To control concurrent insert transactions, under New Data, ensure that Use In Insert is set to Yes.

4. To control concurrent update transactions, under Change Data, ensure that:

   - Use In Update is set to Yes.
   - Expression subproperty for the Use In Update property is empty. Any expression will override Panther default handling of optimistic locking, and transaction processing might produce unpredictable results.
   - In Update Where property is set to No. This causes Panther to automatically handle concurrent update transactions.

     Another method of optimistic locking is to set the In Update Where property to Yes. With this method, the value in the widget is included in the `WHERE` clause of the `SQL UPDATE` statement. However, the widget acting as the version column must have its Version Column property set to No.

5. To control concurrent deletion transactions, under Remove Data, ensure that the In Delete Where property is set to No.

   Another method of optimistic locking would be to set the In Delete Where property to Yes. With this method, the value in the widget is included in the `WHERE` clause of the `SQL DELETE` statement. However, the widget acting as the version column must have its Version Column property set to No.

Notes: Since users will not modify the version number, consider setting the Hidden property (under Identity) to Always.

If used with the transaction manager, the default class setting for the version column is updatable and the styles corresponding to this class are applied.

Examples of Optimistic Locking

The following example, describes the customers table, which has a defined version column:

```
CREATE TABLE customers (
   cust_id        INTEGER     NOT NULL,
   first_name     CHAR (20),
   last_name      CHAR (25),
   phone          CHAR (12),
   version        INTEGER,
   primary key (cust_id) );
```

Inserting Data

In `SQL INSERT` statements, if a widget's Version Column is set to Yes, the database column corresponding to that widget is included in the column list and in the `VALUES` clause. The column's value is automatically set to 1.

When inserting data into the customers table, the generated SQL statement would be:

```
INSERT INTO customers
   (cust_id, first_name, last_name, phone, version)
   VALUES (cust_id, first_name, last_name, phone, 1)
```

Updating Data

In `SQL UPDATE` statements, if the widget's Version Column is set to Yes, the database column corresponding to that widget is added to the `SET` clause and to the `WHERE` clause. In the `SET` clause, the column value automatically increments 1. In the `WHERE` clause, the previous value of the column is listed. Therefore, if someone else has updated or deleted the row, the version column in the `WHERE` clause would no longer match the database value and the statement fails.

The generated SQL statement would be:

```
UPDATE tableName SET columnName = value [ , ... ],
   version-column = before-image-value + 1
   WHERE primary-key = before-image-value
   AND version-column = before-image-value
```

Updating values in the `customers` table, generates the following SQL:

```
UPDATE customers
   SET first_name = first_name, last_name = last_name,
   phone = phone
   WHERE cust_id = cust_id AND version = version
```

Deleting Data

In `SQL DELETE` statements, if the widget's Version Column is set to Yes, the database column corresponding to that widget and its before image value are included in the `WHERE` clause. Therefore, if someone else has updated or deleted the row, the version column in the `WHERE` clause would no longer match the database value and the statement fails.

The generated SQL statement would be:

```
DELETE FROM tableName
   WHERE primary-key = before-image-value
   AND version-column = before-image-value
```

Deleting a record from the customers table, generates the following SQL:

```
DELETE FROM customers
   WHERE cust_id = cust_id AND version = version
```

Viewing the SQL Statements

You can view the statements made by the SQL generator by:

- Choosing Database→Trace On option in test mode. This option is less flexible, but is quicker and is often sufficient.
- Choosing Tools→Generate TM SQL in the editor. For the current screen, the SQL statements that the transaction manager generates for the screen are written to a file.

- Using the debugger. This option provides the greatest flexibility. You can even output the generated SQL statements to a log file.

The examples in this section provide sample SQL and the actual SQL from the SQL generator. These statements were prepared for JDB, and might appear differently for other database engines.

Viewing SELECT Statements

The following example selects rows where the column destination matches a value entered on the client screen:

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
   FROM vacations
   WHERE destination = destination
```

The SQL generator declares a cursor for the `SELECT` statement. The where-condition is specified using a binding parameter (:w0) so that the value is supplied when the cursor is executed, not when it is declared.

```
declare dm_jdb1_19 cursor for select tview1.destination,
   tview1.travel_costs, tview1.hotel, tview1.meals,
   travel_costs+hotel+meals
   from vacations tview1 where ((tview1.destination = :w0))
```

Then, an `ALIAS` statement matches the column name with the widget name. If the widget is not named, the widget number is used in the `ALIAS` statement. The following statement matches widget #1 with the first column in the `SELECT` statement, tview1.destination, etc.

```
with cursor dm_jdb1_19 alias #1, #2, #3, #4, #5
```

Finally, it executes the `SELECT` statement. The value of the binding parameter w0 is set to be data currently in the first occurrence of widget destination.

```
with cursor dm_jdb1_19 execute using w0 = destination[1]
```

Viewing INSERT Statements

The following example inserts rows into both the parent and child table views:

```
INSERT INTO customers (cust_id, first_name, last_name, phone)
   VALUES (cust_id, first_name, last_name, phone)
```

```
INSERT INTO cust_trips (cust_id, destination, paid_flag,
   paid_date)
   VALUES (cust_id, destination, 'Y', paid_date)
```

The SQL generator first declares a cursor for the first `INSERT` statement. The values-list is specified using binding parameters that have a prefix `v_` preceding the column name (for instance, :`v_cust_id`).

```
declare dm_jdb1_18 cursor for insert into customers
   ( cust_id, first_name, last_name, phone )
   values ( :v_cust_id, :v_first_name, :v_last_name,
   :v_phone )
```

Then, the SQL generator executes the `INSERT` statement. The value for the parameter `v_cust_id` is the data currently in the first occurrence of widget `cust_id`.

```
with cursor dm_jdb1_18 execute using v_cust_id = cust_id[1],
   v_first_name=first_name[1], v_last_name=last_name[1],
   v_phone=phone[1]
```

In the `INSERT` statement for the second table view, binding parameters are only needed for three of the columns. The value for the third column is provided by its insert Expression property.

```
declare dm_jdb1_18 cursor for insert into cust_trips
   ( cust_id, destination, paid_flag, paid_date)
   values ( :v_cust_id, :v_destination, 'Y', :v_paid_date)
  with cursor dm_jdb1_18 execute using v_cust_id = cust_id[1],
   v_destination = destination[1], v_paid_date = paid_date[1]
```

Viewing UPDATE Statements

The following example updates a customer's phone number. The desired SQL is:

```
UPDATE customers SET phone = new_phone
   WHERE cust_id = cust_id
```

The SQL generator first declares a cursor for the `UPDATE` statement. The bind parameters for where-condition use the prefix `w_` and the parameters for the `SET` clause use the prefix `s_`. Bind parameters are used so that the values are supplied when the cursor is executed, not when it is declared.

```
declare dm_jdb1_2 cursor for update customers
    set phone = :s_phone
    where cust_id = :w_cust_id
```

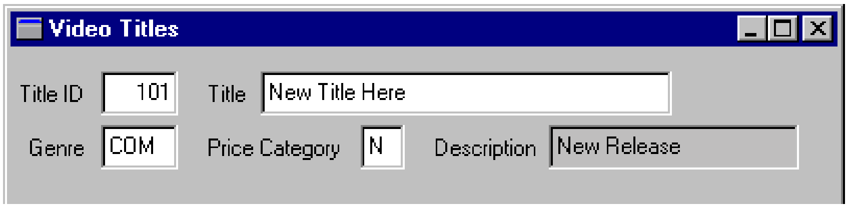
Then, the SQL generator executes the `UPDATE` statement. The value for the parameter `s_phone` is set to be data currently in widget `new_phone`. The values for the parameter `w_cust_id` is in the before image data for this row, indicated by `@bi`. In the following statement, `@bi(#1)[1]` indicates that the parameter's value is in the before image data, from widget #1, in occurrence 1.

```
with cursor dm_jdb1_2 execute using
    s_phone = new_phone[1],
    w_cust_id= @bi(#1)[1]
```

Validating Input Data against the Database

The screen in Figure 33-8 includes a validation link that checks data entry for a valid price category. The Validation Link property on the pricecat widget identifies the link that joins the titles and pricecats table views. At runtime, when a new video title is entered and a valid price category is entered in the pricecat widget, the description of that category is displayed in the `pricecat_dscr` widget. The child table view (in this case, pricecats) is non-updatable, ensuring that only valid data are entered in the titles table.

A validation link enforces a foreign key integrity constraint. When a link widget is to used to enforce a foreign key, the parent table view “references” the child table view. The parent table view is updatable; the child table view must be non-updatable.



Title ID 101 Title New Title Here

Genre COM Price Category N Description New Release

Figure 33-8 A validation link on Pricecat checks for a valid entry and displays information in the Pricecat_dscr data widget.

Implementing a Validation Link

Use a link widget on your screen to specify a validation link for the contents of a data entry widget. The validation link lets a user of your application enter or update data in a widget, and ensures that the SQL generator builds a SQL statement to look up that value in the linked database table. If the value exists, it displays data for any widgets in the child table view. If the value does not exist, it displays an Invalid Entry error.

Specifying a Validation Link

1. Ensure that the link widget identifies the two table views that are joined:
   - The Parent property should specify the table view to which the data entry belongs (the widget that requires validation).
   - The Child property should specify the table view that contains the predefined values.
2. For JetNet and Oracle Tuxedo applications, under Service in the link's Validation Service property, specify the name of the service the transaction manager uses to implement the validation operation.
3. Select the data entry widget that requires the validation link, and under Database in the Validation Link property, specify the name of the link widget.

   The named link is used to validate the data in the selected widget.
4. Ensure that the child table view (named in the Child property of the link widget) has its Updatable property, under Transaction, set to No.

If you want to implement a validation link in an array, set the link's Link Type property to Server. If the Link Type property is Sequential, SQL generation assumes a one-to-many (1:n) relationship instead of a one-to-one (1:1).

Validation Link Processing

Validation link processing is only performed in new and update modes, as part of the `NEW`, `COPY`, `COPY_FOR_UPDATE`, or `SELECT` commands, when entering or updating data. And only after all other widget-level validation have been performed. Therefore, Panther executes the widget validation function and widget-level JPL before it calls the validation link processing.

A foreign key in a table references the primary key columns of another table. For example, the titles table contains the column pricecat; the pricecat column is a foreign key that references the pricecat column in the pricecats table. This ensures that no new value, or nonexistent value, is entered for pricecat in the titles table unless the value already exists in the `pricecats` table.

The first step verifies that the value in the widget (`pricecats`) with the validation link is valid. To test it, the following statement is generated and executed:

```
DBMS DECLARE cursor CURSOR for SELECT 1 FROM pricecats
   WHERE ((pricecats.pricecat == ::l0))

DBMS WITH CURSOR cursor ALIAS @dmtmp

DBMS WITH CURSOR cursor EXECUTE USING l0 = pricecat[1]

DBMS CLOSE CURSOR cursor
```

If the value in the widget `pricecat[1]` exists, `@dmrowcount` is set, and the transaction manager knows the value is valid. The transaction manager deliberately avoids selecting any data to the application, so that the data entry widget is not cleared if no data are found. Instead, it fetches to `@dmtmp`, and does not overwrite the user's data entry.

If the first select is successful, a second select is generated and executed which populates any widgets belonging to the pricecats table view:

```
DBMS DECLARE cursor for SELECT pricecats.pricecat_dscr
   FROM pricecats WHERE ((pricecats.pricecat == ::l0))

DBMS WITH CURSOR cursor ALIAS pricecat_dscr
DBMS WITH CURSOR cursor OCCUR 1 MAX 1
DBMS WITH CURSOR cursor EXECUTE USING l0 = pricecat[1]
```

If the first select fails, Panther displays an Invalid Entry error.

Adding a Lookup to a Validation Link

A validation link might also, optionally, select columns from the referenced table to widgets in the updatable table view. This allows the validation link to supply some suggested values for other widgets belonging to the updatable table view.

Specifying the Lookup

The link's Relations (`rel_op`) property can specify Lookup as well as a Join as a type of relation. With this setting, a widget in the child table view can supply a suggested value for a widget in the parent table view. When you execute a validation link in the transaction manager, the widget in the parent table view is supplied with the suggested data. This suggested value can then be edited, if necessary, in order to save the correct information to the database. The Lookup specification is used only when adding (`INSERT`) and updating (`UPDATE`) database records; lookup relations are ignored when executing `VIEW` and `SELECT` operations.

Note: The parent and the child table views must relate to each other directly, without any table views between them when specifying Lookup as a relations type.

Optionally, you can lookup a child column in the database based on a value the user enters into the widget associated with the Parent column. The lookup is based on the relationship that you define for server or sequential links.

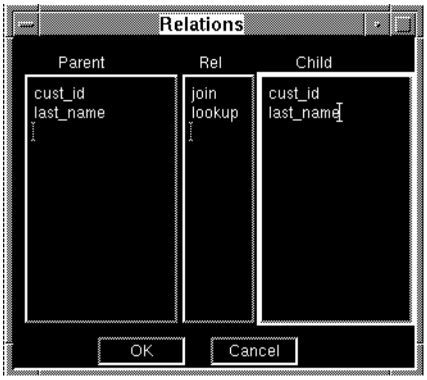
How to Define a Lookup Specification

1. Under Transaction, select the link's Relations property.
2. In the Relations dialog box, specify the parent column name under Parent. If the parent column is represented on the screen by two different widgets, use the widget's name instead. Use the following format (include the square brackets and the literal +0):::`widget_name`[+0]

3. Specify lookup in the center (`rel_opproperty`) column of the Relations dialog box.

4. Specify the Child column name under Child.

5. Select the widget that will receive the data, and under Validation Link, enter the name of the link widget.



Parent	Rel	Child
cust_id	join	cust_id
last_name	lookup	last_name

Figure 33-9 This lookup specification uses the customer id entered by the user to find the corresponding last name in the database table.

For example, consider that the `titles` table contains a `preview_days` column. A value for `rental_days` (number of days video can be rented) is stored in the `pricecats` table, but the video store wants to allow the store manager to alter the number of rental days for very popular new titles without changing the title's price category. The store uses the `preview_days` value to override the default number of rental days. When the store manager enters a new title, the application gives the manager an opportunity to supply a new value (for `preview_days`), but fetches the current `pricecats` value as the suggested value. To support this, the application must modify the Relations (relations) property for the link widget named in the validation link to include a lookup relations type specification:

```
(titles.)pricecat  join  (pricecats.)pricecat
```

should be changed to

```
(titles.)pricecat  join  (pricecats.)pricecat
(titles.)preview_days lookup (pricecats.)rental_days
```

The following SELECT statements are generated to enforce the foreign key:

```
DBMS DECLARE cursor CURSOR for SELECT pricecats.rental_days
   FROM pricecats WHERE ((pricecats.pricecat == ::l0))
```

```
DBMS WITH CURSOR cursor ALIAS preview_days
```

```
DBMS WITH CURSOR cursor OCCUR 1 MAX 1
```

```
DBMS WITH CURSOR cursor EXECUTE USING l0 = pricecat[1]
```

```
DBMS CLOSE CURSOR cursor
```

Instead of fetching to `@dmtmp`, the transaction manager selects the price category's rental days to the `preview_days` widget in the updatable table view `titles`. If it is successful, it continues with the select to populate the non-updatable table view `pricecats`:

```
DBMS DECLARE cursor for SELECT pricecats.pricecat_dscr
   FROM pricecats WHERE ((pricecats.pricecat == ::l0))
```

```
DBMS WITH CURSOR cursor ALIAS pricecat_dscr
```

```
DBMS WITH CURSOR cursor OCCUR 1 MAX 1
```

```
DBMS WITH CURSOR cursor EXECUTE USING l0 = pricecat[1]
```

The store manager can use the suggested value or change the value in `preview_days`. When the title information is saved, the `preview_days` value is saved in the `titles` table, not in the `pricecats` table, where the values remain unchanged.

CHAPTER

34 Specifying Transaction Manager Commands

The transaction manager is controlled by a set of high-level instructions, referred to as *transaction manager commands*, that are called from the application's event functions. After a command is invoked, the transaction manager traverses the table views involved in your application, doing the appropriate processing at each table view it reaches.

This chapter describes:

- How to specify transaction manager commands.
- How to apply a transaction manager command to a portion of the transaction tree.
- How to change transaction modes using the commands.

Transaction Manger Commands

Transaction manger commands (listed in Table 34-1) are used in JPL procedures and C functions (automatically provided by the screen wizard) and are invoked from your application's event functions to execute different types of database operations. For example, there is a command that selects data from the database and a command that clears data from the screen. When a user of your application chooses a command, for instance, via a push button, the transaction manager executes the transaction events associated with the command.

While developing your application with the screen editor, you can access a subset of transaction manager commands via the Transaction menu available in test mode.

A transaction manager transaction must be in progress in order to call a command. A transaction is automatically started on screen entry at which time the transaction manager also verifies that a transaction tree can be built.

Refer to “Screen Entry” on page 17-4 for a list of events that occur when a screen is opened.

Warning: A transaction manager command cannot be called in the screen's unnamed JPL procedure. The transaction manager has not yet been initialized. However, a command can be called as part of a screen entry procedure.

Table 34-1 Transaction manager commands and the associated transaction mode change

Command	Description	Mode change
`CHANGE`	Change to another transaction.	
`CLEAR`	Clear the data from the screen.	
`CLOSE`	Abort the current processing.	initial
`CONTINUE`	Fetch the next group of selected rows (two-tier only).	

Table 34-1 Transaction manager commands and the associated transaction mode change

Command	Description	Mode change
`CONTINUE_BOTTOM`	Fetch the last set of rows using a continuation file (two-tier only).	
`CONTINUE_DOWN`	Fetch the group set of rows using a continuation file (two-tier only).	
`CONTINUE_TOP`	Fetch the first set of rows using a continuation file (two-tier only).	
`CONTINUE_UP`	Fetch the previous set of rows using a continuation file (two-tier only).	
`COPY`	Copy the data currently on the screen, allowing the user to change it in order to enter a new row.	new
`COPY_FOR_UPDATE`	Change the transaction mode to update, allowing the user to change the data currently on the screen.	update
`COPY_FOR_VIEW`	Change the transaction mode to view so the data is for viewing purposes only.	view
`FETCH`	Retrieve one or more rows for a single table view (not recommended).	
`FINISH`	End the current transaction.	
`FORCE_CLOSE`	Abort the current processing.	
`NEW`	Clear the screen so that the user can enter new data.	new
`REFRESH`	Reapply the styles and classes to the screen.	
`RELEASE`	Release transaction manager cursors.	
`SAVE`	Update the database with the new or edited information.	
`SELECT`	Retrieve one or more rows from the database for possible updates.	update
`START`	Start a new transaction.	initial

Table 34-1 Transaction manager commands and the associated transaction mode change

Command	Description	Mode change
`VIEW`	Retrieve one or more rows from the database for viewing purposes only.	view
`WALK_DELETE`	Traverse the transaction tree using the order specified in the Delete Order property.	
`WALK_INSERT`	Traverse the transaction tree using the order specified in the Insert Order property.	
`WALK_SELECT`	Traverse the transaction tree by server view.	
`WALK_UPDATE`	Traverse the transaction tree in using the order specified in the Update Order property.	

Command Syntax

Transaction manager commands are called using the function `sm_tm_command`. The syntax for `sm_tm_command` varies slightly depending upon the command being called. Common syntax is:

```
sm_tm_command ("commandName [ tableView [ tableViewScope ] ]")
```

`commandName`—one of the following (other commands, not included here, use a slightly different syntax):

`CLEAR`	`CONTINUE_UP`	`VIEW`
`CLOSE`	`COPY`	`WALK_DELETE`
`CONTINUE`	`FORCE_CLOSE`	`WALK_INSERT`
`CONTINUE_BOTTOM`	`NEW`	`WALK_SELECT`
`CONTINUE_DOWN`	`SAVE`	`WALK_UPDATE`
`CONTINUE_TOP`	`SELECT`	

Refer to page 8-3 in the *Programming Guide* for syntax and detailed information on all of the transaction manager commands.

`tableView` (optional)—Name of a table view on the screen; it must be the first table view in a server view.

`tableViewScope` (optional)—Generally one of the following values:

- `TV_AND_BELOW`—Applies the command to the specified table view and all server views below it in the tree. This is the default setting if no `tableViewScope` parameter is supplied.
- `BELOW_TV`—Applies the command to the server views below the specified table view.
- `TV_ONLY`—Applies the command to the specified table view only.
- `SV_ONLY`—Applies the command to the specified server view only.

Limiting the Number of Table Views

Most transaction manager commands let you to limit the scope of the transaction by specifying a table view or server view from which to begin the traversal. In this case, the portion of the transaction tree over which the command operates begins with the specified table or server view, rather than the default. If a table view is specified, it must be the first table view in a server view.

Consider the transaction tree in Figure 34-1, the command:

```
sm_tm_command("VIEW tapes")
```

limits the `VIEW` operation to the table view `tapes` and any other table views below it in the transaction tree, in this case, `titles` and `pricecats`.

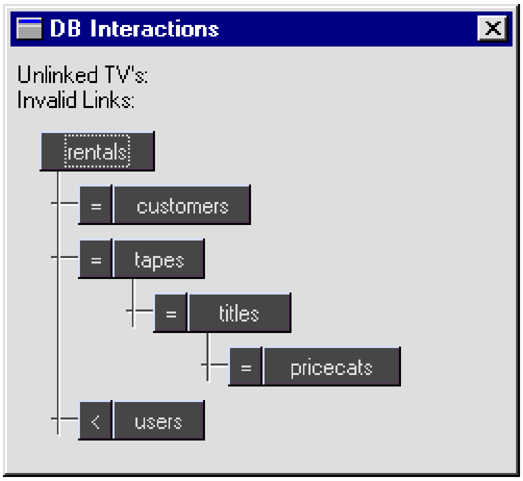


Figure 34-1 Limit the scope of a transaction by identifying the table view in the transaction tree.

Refer to page 31-10 to learn more about how transaction trees are used during event generation.

Implementing Full and Partial Commands

The transaction models use the concept of full and partial commands to determine how to process certain commands. A full command is applied to all table views on the screen and is issued with a single argument, the *`commandName`*. For example:

```
sm_tm_command("VIEW")
```

In this case, the variable `TM_FULL` is set to 1.

A partial command is applied to a limited number of table views and is issued by a command statement that includes a table view or server view parameter. For example:

```
sm_tm_command("VIEW roles TV_ONLY")
```

In this case, `TM_FULL` is set to 0.

In the transaction tree (in Figure 34-1), the following two commands would be equivalent since the partial command specifies the root table view:

```
sm_tm_command("VIEW")
```

```
sm_tm_command("VIEW rentals")
```

In this case, `TM_FULL` is set to 1 for the full command and to 0 for the partial (second) command.

All models use the variable `TM_FULL` to determine when to fully commit a save operation (in two-tier only). Some models use `TM_FULL` to determine the transaction mode after a `SAVE` command.

You can query for the current value of `TM_FULL` using `sm_tm_inquire`.

When executing partial `SELECT` and `VIEW` commands (and related `FETCH` and `CONTINUE` commands), specify the first table view of the server view as the table view. Otherwise, testing for the parent table view is performed on the first table of the server view instead of on a table view at a higher level.

Setting the Transaction Mode

table with mode descriptions also in TMRuntime

When a transaction manager command is executed, the transaction mode for the screen is also defined. The transaction mode defines the protection settings and display attributes of widgets on the screen. For more information about transaction modes and how they give visual and application behavioral cues to users of your application, refer to page 31-14

Table 34-2 lists the transaction modes set by the transaction manager and the commands that initiate those modes.

Table 34-2 Transaction modes and the commands that initiate them

Mode	Description	Command selection
initial	Indicates that no processing is in progress.	`START`, `CLOSE` and `FORCE_CLOSE`

Table 34-2 Transaction modes and the commands that initiate them

Mode	Description	Command selection
new	Allows new data to be entered.	`NEW` and `COPY`
update	Allows existing data to be modified.	`SELECT` and `COPY_FOR_UPDATE`
view	Allows existing data to be displayed.	`VIEW` and `COPY_FOR_VIEW`

Table 34-3 shows which transaction manager commands are available in each mode.

Table 34-3 Transaction manager commands available for each transaction mode

Command	Initial	New	Update	View
`CHANGE`	Y	Y	Y	Y
`CLEAR`	Y	Y	Y	Y
`CLOSE`	Y	Y	Y	Y
`CONTINUE`	N	P	Y	Y
`CONTINUE_BOTTOM`	N	P	Y	Y
`CONTINUE_DOWN`	N	P	Y	Y
`CONTINUE_TOP`	N	P	Y	Y
`CONTINUE_UP`	N	P	Y	Y
`COPY`	Y	Y	Y	Y
`COPY_FOR_UPDATE`	F	F	F	F
`COPY_FOR_VIEW`	F	F	F	F
`FETCH`	N	N	Y	Y
`FINISH`	Y	Y	Y	Y
`FORCE_CLOSE`	Y	Y	Y	Y
`NEW`	F	Y	F	F
`REFRESH`	Y	Y	Y	Y

Table 34-3 Transaction manager commands available for each transaction mode

Command	Initial	New	Update	View
`RELEASE`	Y	Y	Y	Y
`SAVE`	N	Y	Y	N
`SELECT`	Y	Y	Y	Y
`START`	Y	Y	Y	Y
`VIEW`	Y	Y	Y	Y
`WALK_DELETE`	Y	Y	Y	Y
`WALK_INSERT`	Y	Y	Y	Y
`WALK_SELECT`	Y	Y	Y	Y
`WALK_UPDATE`	Y	Y	Y	Y

`Y`=*Available;* `N`=*Not available;* `F`=*Valid for full commands only;* `P`=*Valid for partial commands only.*

CHAPTER

35 Generating Transaction Manager Events

Once a transaction manager command is invoked, the transaction manager generates a series of events.

This chapter describes:

- How transaction manager components process events.
- How the transaction models work.
- How requests and slices are generated.
- How to control the event stack.
- How the transaction manager processes service requests in JetNet applications.

Generating Transaction Manager Events

When a transaction manager command is called, the transaction manager processes the command as a series of events, using information entered in the application screen, as well as information defined in screen properties, and in table view and link widget properties. The control flow of events within the transaction manager (on a three-tier application server or on a two-tier client) is illustrated in Figure 35-1.

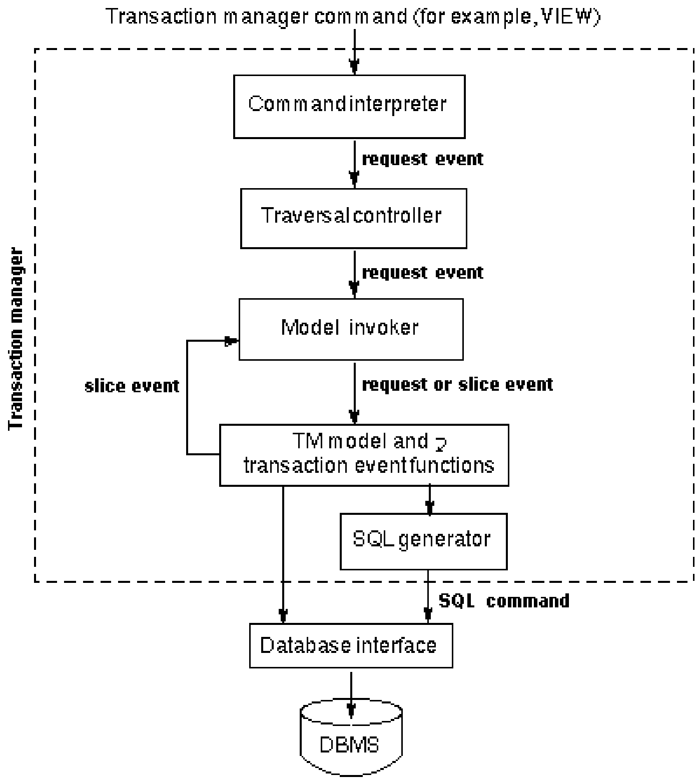


Figure 35-1 The transaction manager processes a command as a series of events.

As shown in Figure 35-1, the events are characterized as either *request* events or *slice* events. When the transaction manager is called, a command interpreter generates a set of request events for the specified command. For several of the more important requests, a sequence of more specialized events, referred to as slices, are generated by the transaction model and/or event functions. Table 35-1 describes this processing sequence for events, requests and slices.

For example, consider the `VIEW` command which fetches database information, the command interpreter generates the following request events for the command:

```
TM_PRE_VIEW
TM_VIEW
TM_POST_VIEW
```

Each request event, for example, `TM_PRE_VIEW`, is sent to the traversal controller, where the request is mapped over the table views for the current screen. When the traversal controller completes the processing for one request, the next request is sent.

Table 35-1 Stages of transaction manager processing

Processing stage	Description	Event type generated
Command interpreter	Generates request events for the specified command. Sends requests to the traversal controller, one at a time.	Requests
Traversal controller	Traverses the transaction tree (derived from the screen's table views and links) for the specified request event. How the tree is traversed depends upon the type of request, the transaction tree it self, and the data on the screen. In general, the traversal controller calls the model invoker to process the request for each table view.	
Model invoker	Invokes an event function (if one is specified) and the appropriate model for the specified table view. For information on how event functions are invoked, refer to page 35-6.	
TM models and event functions	Processes request and slice events for the specified table view. Processing can include calling the SQL generator or database interface, or generating slice events that are routed back to the model invoker and then sent to the model or event function.	Slices

Table 35-1 Stages of transaction manager processing

Processing stage	Description	Event type generated
SQL generator	Generates SQL commands or sends the statement to the database interface.	
Database interface	Interfaces with the DBMS.	

Refer to page 9-3 in the *Programming Guide* for a list of the request and slice events for all transaction manager commands.

Traversing the Table Views

After the command interpreter generates the request events, the next step is to traverse the transaction tree, applying those requests to the database tables that are a part of the screen.

To support automatic database access, Panther needs to know which widgets are associated with which database tables, and how the tables are related. Database table information is stored in a *table view widget*. Text widgets derived from the database table's columns are members of the table view.

The database table's relationship information is stored in *link* widgets; they define the relationship between two table views. Based on these relationships, a transaction tree can be generated. The DB Interactions window, shown in Figure 35-2, illustrates a sample transaction tree for event processing.

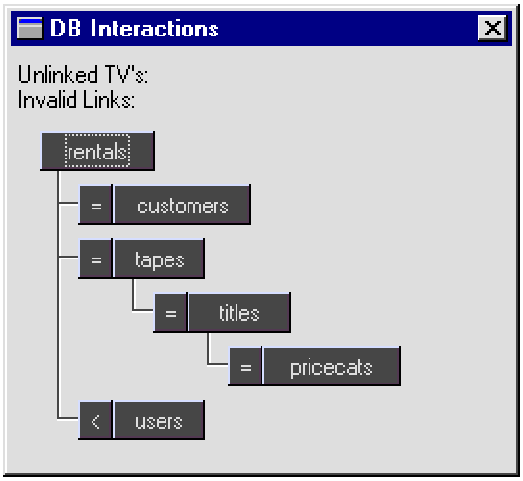


Figure 35-2 The DB Interactions window shows the link relationships between table views on a screen.

Generating Events in the Transaction Model

As the transaction manager traverses the table views, it needs to determine what processing to perform for each event. The processing is defined in event functions and transaction models.

There are two layers of transaction models: the common model and a database-specific model. The common model typically processes most of the events; it provides plausible processing for every event known to the transaction manager and contains the functionality common to all of the database engines. The database-specific model contains processing necessary for a specific database engine.

Invoking Event Functions and Models

Table 35-2 lists the models and event functions that might be invoked for an event (request or slice), and how you invoke functions or specify a particular model.

Table 35-2 How to invoke transaction manager event functions or transaction models

Type	Description	How to specify
Event function	Applied to a specific table view. You can use an event function to affect the processing of either the database-specific model or the common model.	Table view's Function (function) property (under Transaction).
Database-specific transaction model	Applied by default to a table view. Alternatively, a custom model can be identified for a specific table view which would supersede the default database-specific model. You can also modify a distributed model in order to customize its behavior.	Custom model called via table view's or screen's Model (model) property (under Transaction). If a custom model is not specified, a database-specific model is applied.
Common model	Applied to all table views, unless superseded by a database-specific model (default or custom) or event function that processes `TM_OK` event.	Not applicable; called by default on `TM_PROCEED` event

The source code for the database-specific transaction models is provided 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.

Figure 35-3 shows how events are processed with the transaction manager via the model invoker. If an event function is specified for the current table view, then the event function is invoked first. If the event function returns a return code of `TM_PROCEED`, then the invoker also invokes the database-specific transaction model. If the transaction model returns a return code of `TM_PROCEED`, then the common model is invoked.

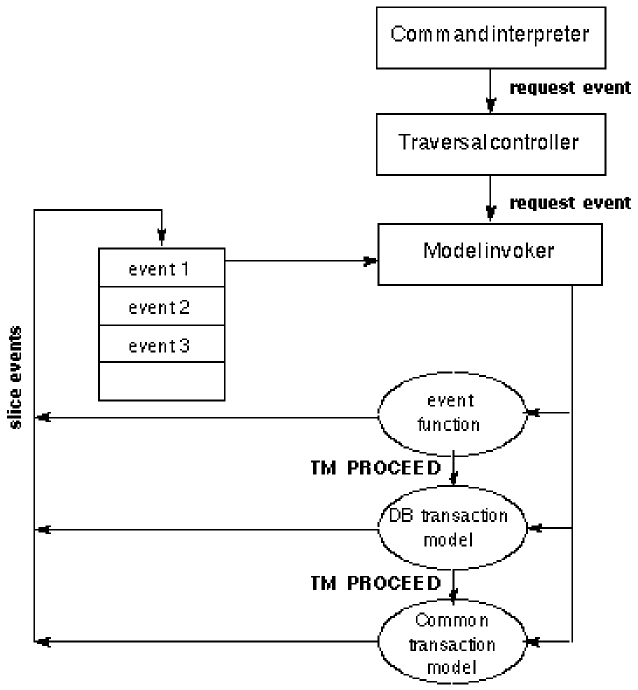


Figure 35-3 Transaction manager event processing and transaction models.

A request event is sent from the traversal controller, that is, as the transaction manager traverses the transaction tree from table view to table view, to the model invoker. The invoker invokes the appropriate event function or transaction model, and passes the request along. The event function or transaction model can then generate slice events for the request and push these events onto the *event stack*. The model invoker then processes the slice events from the stack in a last-in, first-out order.

The slice events are processed in the exact same manner as the original request event, making no distinction between the two. For example, consider the request event `TM_DELETE` which generates several slice events, including `TM_GET_SAVE_CURSOR`. The model invoker does not know which event is a request and which is a slice.

Any slice event can also generate additional slice events, which are then pushed onto the top of the event stack. After all slice events are processed and the stack is empty, processing control returns to the traversal controller, to continue walking the traversal tree, to the next table view.

Event Processing Steps

The invoker uses the following priorities and precedence when processing events:

1. Read the next event from the stack. If the stack is empty, return to the traversal controller.
2. Invoke the event function, if one is specified for the current table view. Otherwise go to step 4.
3. Check the event function return code. If the return value is:
   - `TM_PROCEED`, then continue to next step.
   - `TM_OK`, then event processing is complete. Go to step 1.
   - An error or diagnostic code, then push the relevant error or diagnostic event onto the stack. Go to step 1.
4. Invoke the database-specific transaction model.

   The transaction model can be specified as one of the following and the model invoker determines which model to use based on the order of precedence:

   - Table view's Model property (if specified).
   - Screen's Model property (if specified and if no table view model is specified).
   - Default transaction model (if the table view and screen models are not specified).

   The default transaction model is determined by the target database for that table view. (For example, the default model for JDB databases is `tmjdb1.c`.) For a typical application, one transaction model suffices for all the table views.
5. Check the transaction model return code. If the return value is:
   - `TM_PROCEED`, then continue to next step.
   - `TM_OK`, then event processing is complete. Go to step 1.
   - An error or diagnostic code, then push the relevant error or diagnostic event onto the stack. Go to step 1.
6. Invoke the common transaction model.
7. Check the common model return code. If the return value is:

- TM_PROCEED or TM_OK, then event processing is complete. Go to step 1.
- An error or diagnostic code, then push the relevant error or diagnostic event onto the stack. Go to step 1.

Typically, a few of the events in a transaction can be processed by an event function or database-specific transaction model, while most transaction events are processed by the common model.

Controlling the Event Stack

The following library functions allow you to push and pop events from the stack:

- sm_tm_push_model_event—Place an event on the stack.
- sm_tm_pop_model_event—Remove an event from the stack to prevent it from occurring. sm_tm_pop_model_event returns the event number, or 0 if the stack is empty.
- sm_tm_clear_model_events—Clear the event stack.

Transaction Models and the Event Stack

The following example illustrates how the requests TM_SELECT and TM_VIEW are further subdivided into slices and those slices are pushed onto the event stack in reverse order.

```
case TM_SELECT:
case TM_VIEW:
       /*  Put slices onto the stack only if it is the current
        *  server view
        */

    tv = sm_tm_pinquire(TM_TV_NAME);
    sv = sm_tm_pinquire(TM_SV_NAME);
    if (tv && sv && *sv && !strcmp(tv,sv) )
    {
       sm_tm_push_model_event(TM_SEL_CHECK);
       sm_tm_push_model_event(TM_SEL_BUILD_PERFORM);
       sm_tm_push_model_event(TM_SEL_GEN);
       if (!sel_cursor[0])
       {
          sm_tm_push_model_event(TM_GET_SEL_CURSOR);
        }
    }
```

For a list of the transaction events associated with each command and a description of the processing performed by those events, refer to page 9-3 in the *Programming Guide*.

Adding Your Own Transaction Events

It is possible for you to define your own transaction events and push them onto the stack as long as you specify them correctly and understand how the event stack performs its processing.

All transaction events have an integer associated with them. For user supplied events, the integer must be greater than 2047. The specification of the event can be in a header file or in the transaction model.

The transaction model must also list the event in the case statements at the beginning of the file. Otherwise, the model considers the event to be invalid. As part of that operation, you can write your own function to add to the model. Be sure to track any changes you make to the transaction model since it could change in future Panther releases.

Example

An example of adding transaction events appears in the transaction model for JDB. Since referential integrity is not implemented in JDB, the transaction model checks for duplicate rows when you add new data. This is accomplished by adding two events to the JDB model and calling those events after an insert.

```
#define       DUP_GEN       9901
#defineDUP_BUILD_PERFORM 9902
.
.
.
        case TM_INSERT_EXEC:
        if (check_pkey)
        {
            sm_tm_push_model_event(TM_SEL_CHECK);
            sm_tm_push_model_event(DUP_BUILD_PERFORM);
            sm_tm_push_model_event(DUP_GEN);
        }
        retcode = nsel_exec(EXEC_INSERT);
```

```
break;
case DUP_GEN:
retcode = dup_gen();
break;
case DUP_BUILD_PERFORM:
retcode = dup_build_perform();
break;
```

Logging Transaction Events

As an example, create a log of all the select requests made to your database. You can write an event function (`my_eventLog`) to intercept the slice event `TM_SEL_BUILD_PERFORM`, which is generated by the request `TM_SELECT` and `TM_VIEW`.

```
proc my_eventLog (event)
vars stream, err
{
if event == TM_SEL_BUILD_PERFORM
    {
    stream = sm_fio_open("mylog", "a")
    err = sm_fio_puts("Perform select.", stream)
    err = sm_fio_close(stream)
    }
return TM_PROCEED
}
```

This event function ignores all other events, but when `TM_SELECT` or `TM_VIEW` is encountered, it will log the appropriate information. This event function is applied to the current table view (via its Function property).

The model invoker process the events as follows:

1. `TM_SELECT` or `TM_VIEW` is passed to the model invoker from the traversal controller.

2. The invoker submits the `TM_SELECT` or `TM_VIEW` event to the event function (`my_eventLog`) named in the table view's Function property. The event function performs no processing and sends a return code `TM_PROCEED`.

3. The invoker checks the return code and continues processing.

4. The invoker sends `TM_SELECT` or `TM_VIEW` to the transaction model, which (for this example) also sends a return code of `TM_PROCEED`.

5. The invoker checks the return code and continues processing.
6. The invoker sends `TM_SELECT` or `TM_VIEW` to the common model, which generates several slice events, including `TM_SEL_BUILD_PERFORM` and pushes them onto the event stack. The event `TM_SEL_BUILD_PERFORM` is pushed near the bottom of the stack.
7. The invoker processes the other events on the stack, just as it did for the original request—invoking the event function and then the transaction model—until `TM_SEL_BUILD_PERFORM` moves to the top of the stack.
8. The invoker reads `TM_SEL_BUILD_PERFORM` from the top of the stack and sends it to the event function.
9. The event function recognizes the event `TM_SEL_BUILD_PERFORM` and logs the appropriate information. It sends a return code `TM_PROCEED`.
10. The invoker checks the return code and continues processing the event—first to the database-specific transaction model and then to the common model.
11. The next event is called.

Using the Transaction Model with JetNet/Oracle Tuxedo

On a three-tier client using the JetNet middleware adapter, the transaction manager performs many of the same steps, but it uses the middleware API model, instead of the transaction model, to determine the processing for each event. The middleware API model generates service requests which can then be passed to the middleware for processing.

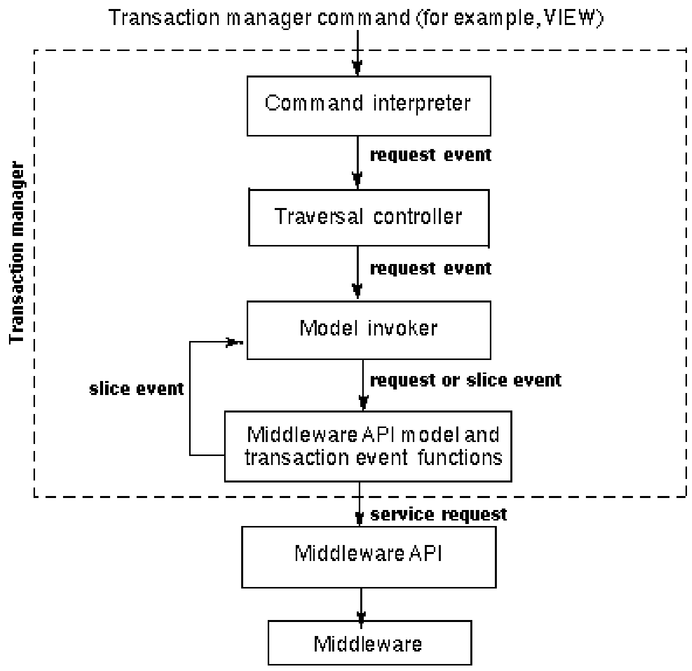


Figure 35-4 A three-tier JetNet client uses the middleware API model.

`jetrb1` is the transaction model provided for three-tier applications using the `JetNet` middleware adapter. When a client uses the transaction manager, `jetrb1` determines which service request to make to satisfy the transaction manager command and provides default processing for the following transaction manager events: `TM_SELECT`, `TM_VIEW`, `TM_DELETE`, `TM_DELETE_EXEC`, `TM_INSERT`, `TM_UPDATE`, `TM_UPDATE_EXEC`, `TM_PRE_SAVE`, `TM_POST_SAVE`, `TM_PRE_CLOSE`, `TM_POST_CLOSE`, `TM_POST_CLEAR`, and `TM_VAL_LINK`.

The transaction manager can then be used on the application server to generate the SQL for the service request.

Figure 35-5 shows how the transaction manager is used in a three-tier application to generate and process a client request.

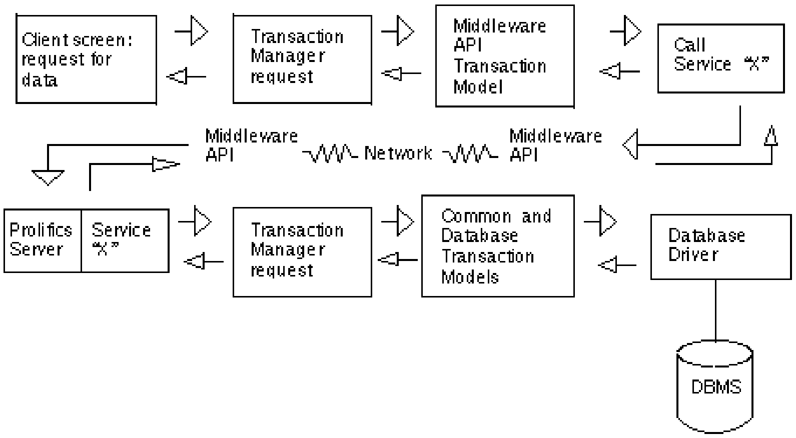

Client screen: request for data
Transaction Manager request
Middleware API Transaction Model
Call Service "X"
Middleware API
Network
Middleware API
Prolifics Server
Service "X"
Transaction Manager request
Common and Database Transaction Models
Database Driver
DBMS

Figure 35-5 The transaction manager can be invoked on both client and server ends of the application.

The application property, `tm_transaction`, determines if a service is transaction-manager enabled and, if so, which transaction manager operation to perform.

For information on using FML buffers with the transaction manager in the middleware adapter for Oracle Tuxedo, refer to page 8-2 in the *JetNet/Oracle Tuxedo Guide*.

CHAPTER

36 Runtime Transaction Manager Processing

This chapter describes transaction manager processing at runtime. It includes:

- How the transaction manager selects data.
- How database updates are performed.
- What runtime properties are available for the transaction manager.
- How to process transaction manager errors.

Running Transaction Manager

Once the application screen is created and its widgets, table views and links are properly defined, the screen is ready to use.

Refer to Chapter 31, “Building a Transaction Manager Screen,” for instructions on building a transaction manager screen.

Opening the Screen

Processing for the transaction manager begins when you open an application screen. On screen entry, the transaction manager automatically executes the following steps:

- Calls the `START` command which assigns a transaction name to this session with the transaction manager.
- Checks the tree traversal of the table views and links to make sure that the root table view can be determined and that there are no circular links.
- Verifies that the functions specified in the table views' Function property are available.
- Sets the screen to initial mode and applies any styles specified for that mode.

If a named function cannot be found or if the root table view cannot be determined, an error message is issued and the transaction manager stops its processing.

Closing the Screen

When you close a screen, the transaction manager performs the necessary exit processing. This includes:

- Calling the `FINISH` command which closes any open cursors and closes the current transaction manager transaction.
- Verifying that the functions specified in the table view's Function property are available, in case they are needed for the `FINISH` command.

If a named function cannot be found, an error message is issued.

Viewing the Generated SQL

The following options are available for viewing the generated SQL:

- On the Database menu, the Trace On option displays the DBMS command or SQL statement generated for each event.
- On the Editor's Tools menu, the Generate TM SQL option generates the SQL for the screen and writes it to a file. These statements can be used as a basis for writing stored procedures.
- The Panther debugger allows you to step through each event.

Disabling the Transaction Manager

If you want to disable the transaction manager, change the screen's Root property to `"-none-"`.

Displaying Data

The transaction manager uses two commands to display data from a database, `VIEW` and `SELECT`. These commands can be invoked whenever a database connection is active.

`VIEW` displays information; `SELECT` allows the user to modify the selected data.

Executing the Select Statement

In general, when a `VIEW` or `SELECT` command is executed, data is fetched from the database and displayed in appropriate widgets using `dbms` commands (as part of the `TM_SEL_BUILD_PERFORM` event):

- `DBMS DECLARE CURSOR`—Creates a named cursor for the `SQL SELECT` statement.
- `DBMS ALIAS`—Maps the column name or select expression to the Panther destination variable. This allows you to have widget names that are different from the database column from which the widgets were derived.

- `DBMS EXECUTE`—Performs the SQL statement associated with the cursor named in the `WITH CURSOR` clause.For example:

  ```
  DBMS DECLARE jdb1 CURSOR FOR SELECT ...
  DBMS WITH CURSOR jdb1 ALIAS ...
  DBMS WITH CURSOR jdb1 EXECUTE USING ...
  ```

This series of dbms statements is performed for each server view, which includes the table view (defined as a server view) and all table views joined to it via server links.

If the Panther targets for the select set are arrays, the first retrieved row populates the first occurrence, the second row populates the second occurrence, and so on.

Figure 36-1 illustrates a screen that lets users enter a video title by name or identification number, and view the names of the actors and their roles for the specified video. In addition, new videos can be added with the corresponding actors and roles.

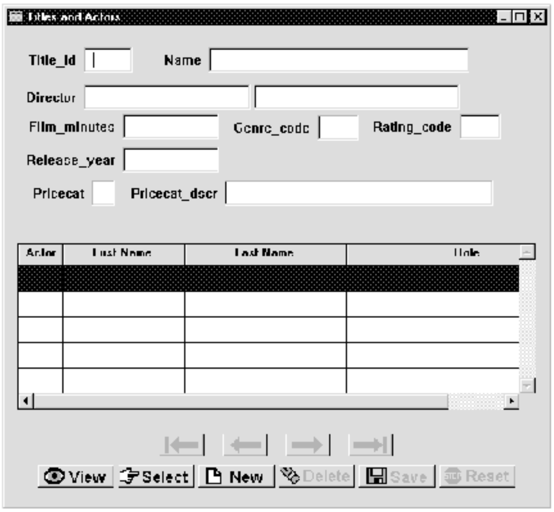


Figure 36-1 Data are displayed by executing a series of DBMS statements.

The screen in Figure 36-1 contains two server views; therefore, the statements are executed twice. The first series fetches a video title and its associated price category from the titles and pricecats tables. The second series fetches the actors appearing in the video and the name of their roles.

Figure 36-2 illustrates the DB Interactions window for this screen. The titles table view has a sequential link with the roles table view; therefore, values in the titles table view (which is the parent) are used to fetch data for the child table view. The Relations property of the sequential link tells the transaction manager which widget to use to supply the value. In this example, the value in `title_id` is used to build a `WHERE` clause which fetches only the actors in that video.

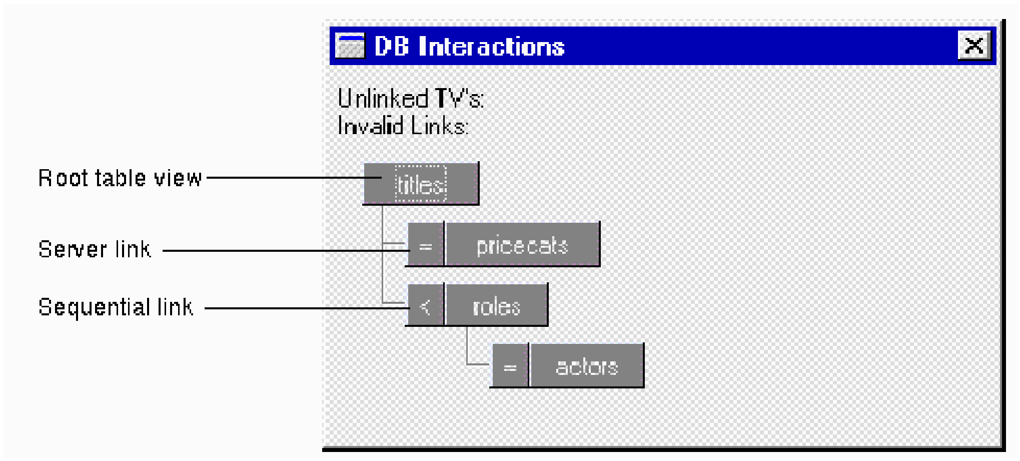


Figure 36-2   The DB Interactions window displays the table views and their associated links.

Scrolling Through the Select Set

The `CONTINUE` command (only available on two-tier processing) in the transaction manager fetches the next set of data for the screen. For the root table view, the next row, or set of rows, is fetched. For any child table views connected by sequential links, additional `SQL SELECT` statements are issued, using the values from the parent table view in the `WHERE` clause. For each subsequent `CONTINUE` command, another set of data is fetched. If there are no additional rows, nothing is done.

There are two ways to allow users to scroll forward and backward through a select set. You can create scrolling widgets or a grid for displaying the data. In environments where memory is limited, you can fetch only a small number of rows to the application and buffer the rest in a disk file. This is known as using a continuation file or a store file.

To use a continuation file with transaction manager, you need to edit the Fetch Directions property for either the screen or the table view. If Fetch Directions is set to Up/Down-all modes or Up/Down-view mode, the transaction manager fetches the data to a continuation file. Then, issuing a `CONTINUE_UP` command displays the previous set of data, and issuing a `CONTINUE_TOP` command displays the first set of data.

Panther does not set backward scrolling via continuation files as the default since Panther does not update the continuation file when the onscreen data is changed. Scrolling backward shows the original, fetched data. If you set Fetch Directions to be Up/Down in all modes, be aware that once a `SAVE` command is issued, you need to re-execute `SELECT` in order to see any updated data.

Controlling the Number of Rows

The number of occurrences fetched for transaction manger is set in the Maximum Occurrences (`max_occurrences`) property.

The Count Select property of each server view allows you to request that the transaction manager count the number of rows in a result set and compare it to a specified threshold value before actually fetching data.

In the server view's properties, set the Count Select property and its subproperty, Count Warning, to Yes. In the Count Threshold property, specify the maximum number of rows to fetch in a result set. If the size of a result set (stored in the server view's runtime `count_result` property) exceeds this value, the user is prompted before the data is actually fetched.

For any command which can modify the database, such as `SELECT`, the transaction manager must synchronize the widgets in a server view. For more information, refer to page 36-8, “Updating Data in Arrays.”

Customizing Select Processing

In order to customize the processing done for any table view, you can:

- Write a transaction event function for a request or slice event and specify the name of the function in the table view's Function property. (See Chapter 32, “Writing Transaction Event Functions.”)

- Modify the SQL Generation properties for widgets, table views, or links. (See Chapter 33, “Using Automated SQL Generation.”)
- Specify the system for selecting data in the Method property:
  - SQL Statement Generation— (default) The transaction manager generates SQL statements based on the current property settings.
  - Function Call—Specify the JPL procedure or C function which will perform select processing.
  - Nothing—Do not perform any select processing for this table view.
- Call the functions which modify the generated SQL statements. (See page 33-31, “Modifying SELECT Statements.”)

Updating the Database

If you execute a command that modifies data, such as `NEW`, `SELECT`, `COPY` or `COPY_FOR_UPDATE`, the transaction manager initiates before-image processing for all updatable table views. Panther's before image remembers the original values of the fetched data. Then, when you execute `SAVE`, the transaction model generates the necessary statements so that the database matches the current data on the screen.

The `NEW` command prepares the screen for data entry; it does not insert information into the database. Similarly, the `SELECT` command retrieves data from the database and prepares the screen to be edited by changing the screen to update mode. The changes made on the screen are not saved to the database until a `SAVE` command is executed.

Traversal for Database Updates

Specialized traversal patterns are used for the `TM_INSERT`, `TM_UPDATE`, and `TM_DELETE` requests. For these requests, the traversal pattern is determined by the link properties for the transaction tree and the operation being performed.

You can change the traversal order for the TM_INSERT, TM_UPDATE, and TM_DELETE requests by modifying the link widget's Insert Order (insert_order), Update Order (update_order), and Delete Order (delete_order) properties, indicating whether the parent (PV_PARENT_FIRST) or child (PV_CHILD_FIRST) table view should be processed first.

Updating Data

The transaction manager SELECT command queries the database for information so that it can be updated. When you execute the SELECT command, the transaction manager fetches the first screenful of data for each of the linked table views. (If you then execute the CONTINUE command, the transaction manager fetches the next screenful of data.)

Panther keeps track of the changes the user makes while the screen is in update mode. When the application executes SAVE, the transaction manager generates the statements to update the database. If the application attempts any other transaction manager operation, the transaction manager prompts the user if it should discard the changes. If the user chooses to discard, the transaction manager proceeds to the next command without changing the database. If the user chooses not to discard, transaction manager returns control to the screen. You can modify this behavior if you wish.

When before image processing is activated, each time a field is modified in some way (data is edited, data is cleared, new data is entered), the data previously in the widget is copied into memory and the transaction manager is notified that data on the screen has changed. Then, when the SAVE command is selected, the transaction manager looks at the changes and determines which statements are necessary to update the database so that it matches what is currently on the screen.

Updating Data in Arrays

For any command which can modify the database, such as SELECT, NEW, COPY and COPY_FOR_UPDATE, the transaction manager must synchronize the widgets in a server view. This ensures that any updates occur on the same occurrence of each widget in the server view. Each time the SELECT, NEW, COPY and COPY_FOR_UPDATE commands are chosen, the transaction manager attempts to synchronize the widgets in a server view if:

- The table view is updatable.

- The widgets' Synchronization property is set to Default or Yes.
- If the Synchronization property is set to Default, then if the widgets' Use In Select, Use In Insert, and Use In Update properties are set to Yes.

If you get a synchronization error, check to see if the widgets can be set to the same number of occurrences. If this is not possible, review the Use In Select, Use In Insert, and Use In Update properties for each widget to see if they can be changed. Another property change you can make is setting the Synchronization property to No.

However, changing the Synchronization property to No does not change the way Panther fetches data to arrays. The number of rows fetched from the database equals the least number of occurrences set for any widget in the server view whose Use In Select property is set to Yes.

Using Multi-text Widgets

For multi-text widgets with the Word Wrap property set to Yes (the default setting), you need to set the Maximum Occurrences (`max_occurrences`) property.

Changing the Primary Key

If you change value of a primary key, the transaction manger first deletes the row associated with the old value and then inserts a row with the new value.

You can change this behavior by setting the application property `primary_key_update` to Yes which performs a `SQL UPDATE` for the primary key change. Deleting the row can lead to data loss when there are database columns that are not in the table view.

In both cases, if you change the value of a primary key to a null value, the transaction manager deletes the row.

Deleting Data

To delete data in the transaction manager, execute the `CLEAR` command, followed by `SAVE`. This removes all the data displayed on the screen from the database. The `TM_DELETE` request usually processes child table views first. The resulting `SQL DELETE` statement contains a `WHERE` clause built from the before image values of the primary key widgets.

The user can also use the logical key `DELL` to delete a line. Since the transaction manager synchronizes the arrays in a server view, using this key deletes the same occurrence in every array in the server view. You can program a delete line event by calling the function `sm_doccur` or `sm_1clear_array`.

Clearing Data in Arrays

For the `CLEAR` command, the transaction manager clears the fields in each table view (by calling `sm_1clear_array`). To have the transaction manager clear the fields in each server view (the equivalent of calling `sm_clear_array`), you need to either:

- Set the runtime property `tm_clear_fast` to `PV_YES`.
- Call the `dm_set_tm_clear_fast` function.

However, setting `tm_clear_fast` to Yes can be over-inclusive, clearing too much data, or under-inclusive, clearing too little data. In particular, widgets in the synchronization group that do not belong to the table view will get cleared.

If widgets have been excluded from synchronization (via the synchronization property) or if widgets are members of a non-updatable table view, they will not be cleared.

If `tm_clear_fast` is set to Yes, it is not recommended that you call `CLEAR` before the synchronization of tables views occurs.

Do not change this property 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.

Inserting Data

To insert data in the transaction manager, execute the `NEW` command, let the user complete the data entry, and then execute `SAVE`. This inserts a row in each table view that was modified on the screen.

If the data is in arrays or grids, the user can use the logical key `INSL`, which inserts a line. Since the transaction manager synchronizes the arrays in a server view, using this key inserts a line in each array in the server view. You can program an insert line event by calling the function `sm_ioccur`.

For inserts, the transaction models call the SQL generator to build a values list for the widgets in the table view whose Use In Insert property is set to Yes.

Saving Data

Since the `SAVE` command generates different types of statements, depending on whether it needs to insert, update or delete data, the transaction manager checks the value of the variable `TM_OCC_TYPE` to determine what kind of change was made to the row or occurrence. (Refer to "Determining How Screen Data Has Changed" on page 36-26 for a description of `TM_OCC_TYPE` values; the values are defined in `tmusubs.h`.)

To check the status of changes generated by the `SAVE` command, you can call `sm_tm_inquire`(`TM_SAVE_COUNT`) which returns the number of rows that were saved to the database.

Note: The value returned is not equivalent to the number of SQL statements issued, since multiple SQL statements can be issued for each row.

If at any time in this process, you wish to abort the edits to the screen, you can execute the `CLOSE` command which discards the user's changes and puts the screen in initial mode.

Customizing Database Updates

In order to customize the processing done for any table view, you can:

- Write a transaction event function for a request or slice event and specify the name of the function in the table view's Function property. (See Chapter 32, "Writing Transaction Event Functions.")
- Modify the SQL Generation properties for widgets, table views, or links. (See Chapter 33, "Using Automated SQL Generation.")
- Specify the system for selecting data in the Method property:
    - SQL Statement Generation—(default) The transaction manager generates SQL statements based on the current property settings.

      For update and insert processing, the Regenerate SQL sub-property specifies whether the transaction manger should include every database column in the SQL statements or only the columns that have changed.

This property, named `regenerate_ins_sql` and `regenerate_upd_sql` at runtime, can be set at the table view level or at the application level. Table views, by default, use the application level property value.

- Function Call—Specify the JPL procedure or C function.
- Nothing—Do not perform any database updates for this table view.

Transaction Modes

The transaction manager uses a set of transaction modes to help monitor and control your application's appearance and behavior. Transaction modes are initiated by associated transaction manager commands.

At a given point in time, the transaction mode determines:

- Which transaction manager commands are available.
- The behavior of the widgets on your screen.

For example, in a typical client screen the Save button is disabled when you first enter the screen. In this case, the initial transaction mode causes the Save button to be disabled (grayed) and precludes use of the `SAVE` command.

Table 36-1 lists the transaction modes set by the transaction manager and the commands that initiate those modes.

Table 36-1 Transaction modes and the commands that initiate them

Mode	Description	Command selection
initial	Indicates that no processing is in progress.	`START, CLOSE and FORCE_CLOSE`
new	Allows new data to be entered.	`NEW and COPY`
update	Allows existing data to be modified.	`SELECT and COPY_FOR_UPDATE`
view	Allows existing data to be displayed.	`VIEW and COPY_FOR_VIEW`

The following commands do not change the transaction mode, but are available in certain modes:

- `CONTINUE` (and its variants) and `FETCH` are available in update or view modes.
- `SAVE` is available in new or update modes.
- `CLEAR`, which clears the screen, is available in all modes and has no effect on the mode setting.

When you execute a command, the transaction manager checks the current transaction mode and either changes the mode or reports an error if the current mode is invalid for the command. Each screen can have its own mode at any given time.

In addition, when the mode changes, the appearance and protection of widgets can also change depending on the style and class settings for the widget.

Transaction Styles and Classes

As some of the transaction commands are executed, widget behavior is automatically affected. For instance, data entry type widgets can prevent input, menu choices might be deactivated, and push buttons can go from gray (inactive) to becoming accessible. These changes occur because predefined style and class settings exist for each transaction mode.

The style and class settings give a consistent user interface to an application. Data entry widgets can have the same focus and protection settings; the same color changes. This takes place without having to write any source code or set any properties in the screen editor.

The definitions for the styles and classes are defined in the `styles.sty` file in the distributed `client.lib` and, for JetNet /Oracle Tuxedo applications, in the distributed `server.lib`. You can use the predefined styles and classes, or edit style and classes settings or define your own styles and classes using the styles editor (for details on using the styles editor, refer to page 23-1 in the *Using the Editors*).

Applying Styles

In general, when a `SELECT` command is executed, rows from the database are retrieved for possible edit, and the transaction mode is set to update. On the screen illustrated in Figure 36-1 on page 36-4, the `title_id` and `actor_id` widgets are automatically protected from input, preventing any edits to primary key fields. The `pricecat_dscr`, `first_name`, and `last_name` widgets are also protected since they members of non-updatable table views. The remaining data entry widgets can be updated.

For a given transaction mode, certain types of widget behavior are determined by:

- The widget's class property setting, and
- The transaction style, defined by a styles widget (created and edited via the styles editor).

Therefore, a widget's class property specification determines the style, that is, the behavior/appearance of that widget given a particular mode.

Default transaction styles

A transaction style defines a limited set of widget properties. Table 36-2 lists the property settings for the default transaction styles.

Table 36-2 Property settings associated with transaction styles

Style	Property Settings
change	Allows focus, input, clearing, and validation
edit	Allows focus, input, clearing, and validation
show	Prevents focus, input, and clearing; allows validation
visit	Allows focus and validation; prevents input and clearing

In addition, there are other predefined styles to use with menu selections and push buttons. For more information on these styles, refer to page 23-6 in the *Using the Editors*.

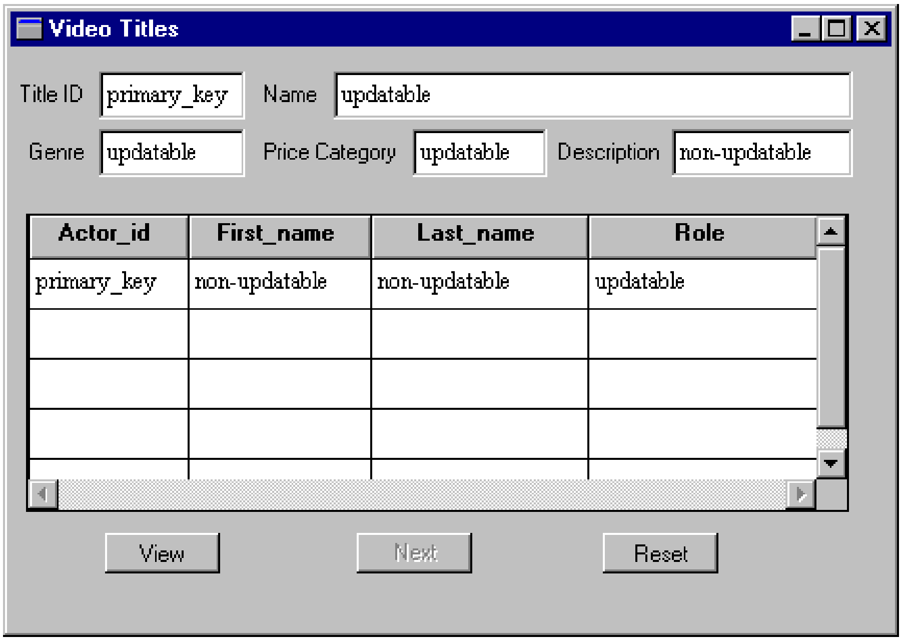
Default transaction classes

Each of a default transaction classes already has a style assigned to each of the transaction modes. If you specify a new class, you must assign a style to each of the modes.

Table 36-3 lists a description for the default transaction classes and the style assignments in each mode. To see how these transaction classes are applied to the data widgets in the sample screen, refer to Figure 36-3.

Table 36-3 Transaction classes and their style assignments

Transaction class	Description	initial	new	view	update occ	update
`non-updatable`	Widget can only be entered in initial mode. Data cannot be edited in any mode if the widget is a part of a non-updatable table view.	edit	show	visit	show	show
`primary key`	Data cannot be edited in update or view modes if the widget is part of the table's primary key in an updatable table view.	edit	edit	visit	visit	edit
`updatable`	Data can be edited except in view mode if the widget is a member of an updatable table view and is not part of the table's primary key.	edit	edit	visit	change	edit



Title ID	primary_key	Name	updatable		
Genre	updatable	Price Category	updatable	Description	non-updatable

Actor_id	First_name	Last_name	Role
primary_key	non-updatable	non-updatable	updatable

Figure 36-3 A sample screen with the default transaction classes.

In the sample screen, the default class assignments are as follows:

- primary key: `title_id` and `actor_id`
- non-updatable: `first_name`, `last_name`, and `pricecat_dscr` are members of table views where the Updatable property is set to No.
- updatable: all remaining widgets is updatable.

You can change the class of any widget by editing the Class property.

Accessing Transaction Information

When you customize transaction manager behavior, you might need to access various property settings or obtain information about the current transaction. Information is available via:

- Library functions
- Transaction manager variables
- JPL access to property values

Using Functions to Set Transaction Manager Behavior

Several library functions are used by the transaction manager, but two of the functions are provided specifically to obtain information about the current transaction—`sm_tm_pinquire` and `sm_tm_inquire`. For example, the name of the current transaction and the root table view of the current transaction are two of the transaction attributes. Some of these attributes can be set by calling the functions `sm_tm_pset` and `sm_tm_iset`. For a complete list of the attributes, refer to the functions `sm_tm_pinquire` and `sm_tm_inquire`.

Using Transaction Manager Variables

Table 36-4 lists the transaction manager variables available for transaction event functions written in JPL.

Table 36-4 Transaction manager variables for writing transaction event functions

Variable	Description	Availability
`@bi(field)[occurrence]`	Access the before image value of the specified field and occurrence. `field` can be the widget name or field number. The variable `@tm_occ` can be used to specify the current occurrence.	In an event function during `TM_DELETE_EXEC`, `TM_INSERT_EXEC`, or `TM_UPDATE_EXEC` events.
`@tm_occ`	Occurrence number being processed. Equivalent to `sm_tm_inquire` (`TM_OCC`). Negative value indicates a deleted occurrence.	In any event function
`@tm_occ_type`	Code reflecting the change, if any, from its before image. Equivalent to `sm_tm_inquire`(`TM_OCC_TYPE`). Refer to page 36-26 for the code values.	In any event function
`@tm_pocc`	Parent occurrence number. Equivalent to `sm_tm_inquire` (`TM_PARENT_OCC`).	In any event function
`@tm_save_cursor`	Name of the cursor used for non-`SELECT` statements. Equivalent to `sm_tm_pinquire` (`TM_SAVE_CURSOR`).	In any event function
`@tm_sel_cursor`	Name of the cursor used for `SELECT` statements. Equivalent to `sm_tm_pinquire` (`TM_SV_SELECT_CURSOR`).	In any event function
Results are unpredictable if these variables are called outside of a transaction event function.		

Example of using transaction manager variables

The following event function performs a “logical” delete on a database row. Instead of physically removing the row from the database, the two slices from the `TM_DELETE` event, `TM_DELETE_DECLARE` and `TM_DELETE_EXEC`, mark the row as deleted so that it can be excluded from selection.

```
proc logicalDeleteEvent (event)
{
    if (event == TM_DELETE_DECLARE)
    {
        if (@tm_occ < 0)
        {
            DBMS DECLARE :@tm_save_cursor CURSOR FOR \
                UPDATE customers \
                SET deleted = 1 \
                WHERE cust_id = ::p1
            return TM_CHECK
        }
    }
    else if (event == TM_DELETE_EXEC)
    {
        if (@tm_occ < 0)
        {
            DBMS WITH CURSOR :@tm_save_cursor \
            EXECUTE USING @bi(cust_id)[:@tm_occ]
            return TM_CHECK_ONE_ROW
        }
    }
    return TM_PROCEED
}
```

In this example, the customers table view includes a column named `deleted`. The column is used to flag the record so that it is excluded from selection. Also, the use of `@tm_save_cursor` supplies the cursor name, `@bi` obtains the before image value for `cust_id`, and `@tm_occ` supplies the occurrence number.

Using Traversal Properties

In addition to the properties available through the screen editor's Properties window for each object, there are properties associated with the transaction manager which contain information about the current traversal tree. These properties, known as traversal properties, are accessible programmatically using JPL. All transaction manager related properties are readable at runtime, and some of them are settable as long as they are not currently participating in transaction.

The traversal properties provide information about the application (or current transaction), server views, table views, links, and widgets. Some of the properties can apply to more than one object type and return different information depending on what is specified. For example, the property sv, when specified for the application with `@app()->sv`, returns the name of the server view participating in the current transaction. When specified for a widget, for example `title_id->sv`, it returns the name of the server view associated with the widget `title_id`.

To use traversal properties effectively, you need to know when a table view is also a server view. A table view is defined as a server view if:

- A sequential link connects the table view to its parent.or
- It is the root table view. To determine the root table view of the current transaction, call `sm_tm_pinquire(TM_ROOT_NAME)`. The root property is a screen property and does not necessarily describe an active transaction.

Many of these properties allow you to programmatically iterate through all occurrences of widgets or table views that are participating in the current transaction or all occurrences of widgets belonging to a particular table view. In addition, many properties identify an object (widget, table view, or link) by returning its ID. Panther converts the ID to a string if one is provided; for example, a widget's field number is returned if its Name property has no value.

If a table view or link is not in the current traversal tree, no information is available. If this is specified in a JPL procedure, you receive the message:

```
Bad field name, #, or subscript.
```

Table 36-5 lists application-level traversal properties that are available for identifying, at runtime, the participants associated with the root table view of the current transaction.

Table 36-5 Application traversal properties

Property	Description
`field_below[int]`	Identifies widgets participating in current transaction.
`num_fields_below`	Number of widgets participating in current transaction.
`num_svs_below`	Number of server views participating in current transaction.
`num_tvs_below`	Number of table views participating in current transaction.
`sv`	Root table view of current transaction.
`sv_below[int]`	Names of server views participating in current transaction.
`tv_below[int]`	Names of table views participating in current transaction.

Reading the Current Transaction

The DB Interactions window, accessible via the screen editor, can illustrate how the you can monitor and control the processing of table views and server views via runtime access to properties through JPL.

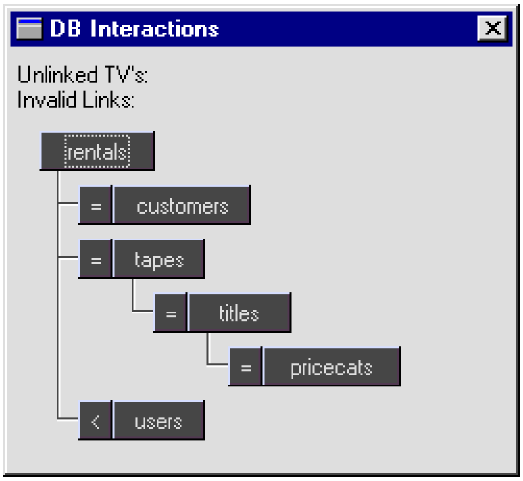


Figure 36-4 DB Interactions window illustrates the tree of table views, server views, and their links.

Figure 36-4 shows the current screen having three server views:

- `rentals`, because it is the root table view, and
- `users` and `titles`, because they each have sequential links to their respective parent table views.

The property `num_svs_below` provides the number of server views at and below the specified server view. Therefore, if `rentals` is specified, the `num_svs_below` value is 3 and the `sv_below` property can provide the names of the server views: `rentals`, `users`, and `titles`. If `customers` is specified, the `num_svs_below` property value returned is 1 even though `customers` is part of a server view which has server views below it, `customers` itself is not a server view and has no children with sequential links. The 1 is returned for the `customers` table view itself.

In general, to identify the objects in transaction, that is, the objects participating in the traversal tree of the transaction that contain a specified table or server view, you might determine the number of objects in the traversal tree prior to seeking the identity of a particular object.

Getting and Setting Property Values via JPL

The following JPL procedure illustrates how, on field entry, to determine to which server view the current field is a member. Given that, the procedure executes the transaction manager `VIEW` command for that specific server view.

```
proc get_sv_query

if K_ENTRY
{
    vars value1
    value1 = name->sv
    call sm_tm_command("VIEW :value1")
}
return
```

If you use server view-specific properties to refer to a table view that is not a server view, no error is reported. Instead, it returns the information for that table view; for example if you use `num_svs_below` on a table view that is not a server view, instead of returning the number of server views that participate in the current server view, the value returned is 1 which is derived from the table view itself.

Table 36-6 Table view traversal properties

Property	Definition
`bi_status[int]`	Reports statuses (fetched data, changed data, or empty) of rows or occurrence in specified table view.
`child[int]`	Names of child table views (from `1` to `num_children`) in specified table view.
`field[int]`	Identifies widgets (from 1 to `num_fields`) in specified table view.
`field_below[int]`	Identifies widgets (from 1 to `num_fields_below`) in specified server view and in its child table views.

Table 36-6 Table view traversal properties *(Continued)*

Property	Definition
`key_constant[int]`	Identifies primary key constants (from 1 to `num_key_columns`) in specified table view (updatable table views only, after `SELECT`, `NEW`, `COPY` or `COPY_FOR_UPDATE` command).
`key_field[int]`	Identifies primary key fields (from 1 to `num_key_columns`) in specified table view (updatable table views only, after `SELECT`, `NEW`, `COPY` or `COPY_FOR_UPDATE` command).
`num_children`	Number of child table views in specified table view.
`num_columns`	Number of columns in the specified table view, derived from the number of occurrences of the Columns property.
`num_fields`	Number of widgets in specified table view.
`num_fields_below`	Number of widgets in specified table view and in its child table views.
`num_key_columns`	Number of widgets that comprise the primary key (indicated in Primary Key property) of the specified table view (updatable table views only, after `SELECT`, `NEW`, `COPY` or `COPY_FOR_UPDATE` command.
`num_sorts`	Number of widgets in the specified table view that have a `sort_widgets` property specifications.
`num_sv_fields`	Number of widgets in specified server view (as identified by the sv property) or in direct or indirect child table views connected via server links to the specified server view.
`num_svs_below`	Number of server views connected to specified server view; the number includes the specified one and all direct and indirect child table views connected to parent table views via sequential links.

Table 36-6 Table view traversal properties *(Continued)*

Property	Definition
`num_tvs`	Number of child table views connected directly or indirectly, via server links, to the specified server view (as identified by the sv property).
`num_tvs_below`	Number of table views, including the specified server view, that are direct or indirect child table views of the specified server view.
`parent`	Name of parent table view (if any) of specified table view. There is at most one parent to any table view; the root table view of a transaction has no parent.
`parent_link`	Name of link in which specified table view is the child. The root table view of a transaction has no parent.
`source_occ`	Occurrences in server view that were valid when child table view was fetched.
`sv`	Name of server view containing specified table view.
`sv_below [int]`	Name of specified server view and all direct or indirect child server views (from 1 to `num_svs_below`) to which it is connected.
`sv_field[int]`	Identifies widgets (from 1 to `num_sv_fields`) in specified server view.
`tv[int]`	Name of specified server view and all table views connected directly or indirectly (from 1 to `num_tvs`) via server links to the specified server view.
`tv_below[int]`	Names of table views connected directly or indirectly (from 1 to `num_tvs_below`) to the specified table view.

Identifying a Widget's Table View

A widget must be a member of a table view in order to participate in a database transaction in order to get values for its traversal properties. The properties listed in Table 36-7 return the name of the table view to which the widget is a member.

The term server view is defined as the table view containing the specified widget and is considered to be a server view if it is joined to its parent table view by a sequential link. If the table view not a server view, the value returned is the closest table view, in the chain of successive parent table views of the transaction above the specified widget, that is not joined to its parent table view by a server link. (The root table view of the transaction, since it has no parent, is never joined to a parent by a server link. Every widget having membership in a table view in a transaction is a member of some server view.)

Table 36-7 Widget traversal properties

Property	Definition
`sv`	Name of server view containing specified widget.
`tv`	Name of table view containing specified widget.

Identifying Links

To identify the link between two table views, use the `parent_link` property, which identifies the link that connects a specified table view with its parent table view. Once you identify of the link, you can obtain its property definitions.

Table 36-8 Link traversal properties

Property	Definition
`rel_child`	Identifies a column belonging to the child table view with which the link connects to a column belonging to the parent table view.
`rel_op`	Identifies type of relationship between the parent and child components of the link: `PV_JOIN` (refer to page 33-22 for details on joining two database tables) or `PV_LOOKUP` (refer to page 33-46 for details on defining a lookup for validation links)
`rel_parent`	Identifies a column belonging to the parent table view with which the link connects to a column belonging to the child table view.

Table 36-8 Link traversal properties

Property	Definition
`num_relations`	Number of entries defined in the relations property. Each entry defines the relationship between a parent and child table view.

JPL Properties for Transaction Manager Operations

There are additional application properties for other aspects of transaction manager operations.

Table 36-9 Transaction manager application properties

Property	Definition
`primary_key_update`	Generate a `SQL UPDATE` statement for primary key changes (instead of a `SQL DELETE` and `INSERT`).

In JetNet/Oracle Tuxedo applications, additional application properties are available:

Table 36-10 Service request application properties

Property	Definition
`tm_transaction`	Determine whether a service is transaction manager-enabled, and if so, which transaction manager operation is to be performed.

Determining How Screen Data Has Changed

Use the function `sm_tm_inquire` to determine the current value of `TM_OCC_TYPE`. The transaction models use the `sm_bi_compare` function to query for the type of change made to a row or occurrence. The values of `TM_OCC_TYPE` are:

`BI_DELETED`	Occurrence was deleted.

`BI_INSERTED`	New data was entered.
`BI_KEY_CHANGED`	Primary key was edited. The before image and the current value of a key field are different.
`BI_KEY_NULLED`	Primary key was changed to `NULL`.
`BI_UPDATED`	Data was updated. The before image and the current value of a non-key field are different.
`BI_UNCHANGED`	No changes were made.
`BI_UNDETERMINED`	Error occurred since change was undetermined.

Processing Errors in the Transaction Manager

A transaction manager command is composed of a series of transaction events. If an error occurs while processing an event, the `TM_STATUS` variable is set to a value other than zero. When the transaction manager completes all the event processing for a command, it then displays an error message.

This section includes information about:

- How to determine if an error has occurred by using the value of `TM_STATUS`.
- What happens to a transaction event when an error occurs.
- Controlling error messages issued by the transaction manager.

For a listing of common transaction manager errors, refer to page 10-1. For information about how event functions deal with errors, refer to page 32-4.

Identifying the Value of the TM_STATUS Variable

You can test for the current value of `TM_STATUS` by using the function `sm_tm_inquire`(`TM_STATUS`).

The default processing by the transaction manager sets `TM_STATUS` to `-1` if there is an error, but you can define other non-zero values for errors if needed.

Generally, the transaction manager sets `TM_STATUS` to a non-zero value only if its current value is zero. In this way, the value set by the first error is not overwritten by errors in later events.

Setting the Value of TM_STATUS

`TM_STATUS` can be set to return a certain value in all conditions or only to return that value if its previous value is zero.

A transaction model or event function can set the return value unconditionally by calling:

```
sm_tm_iset(TM_STATUS, return_value)
```

However, the database-specific models set the `TM_PROPOSE_STATUS` parameter:

```
sm_tm_iset(TM_PROPOSE_STATUS, return_value)
```

This sets the `TM_STATUS` variable only if the previous value was zero. This prevents a non-zero return value resulting from a previous error from being overridden. The transaction models also use this after calling the function `dm_dbms` to execute SQL statements. This preserves non-zero values that might have been set by a error handler (if some other non-zero value had not already been set in `TM_STATUS`).

Event Processing after Errors

Most transaction manager commands are subdivided into three transaction requests:

```
TM_PRE_commandName
```

```
TM_commandName
```

```
TM_POST_commandName
```

For example, the `VIEW` command is divided into three requests: `TM_PRE_VIEW`, `TM_VIEW`, and `TM_POST_VIEW`.

Generally, an error does not prevent the processing of the `TM_PRE_` and `TM_POST_` requests, but does prevent processing of the “main” request. Even if an error occurs in `TM_PRE_` or `TM_POST_` request processing, it does not interfere with table view traversal. However, traversal for the main request ceases when an error is encountered, and never even begins if an error is encountered in `TM_PRE_` processing. Traversal for `TM_POST_` processing begins immediately after main processing ceases.

Consider the processing of the three requests associated with the `VIEW` command. Normally, each table view processes `TM_PRE_VIEW`. Next, each table view processes `TM_VIEW`. Then, each table view processes `TM_POST_VIEW`. If an error is encountered during `TM_PRE_VIEW`, then processing continues to the `TM_POST_VIEW` request, but no `TM_VIEW` processing is done. In particular, this allows `TM_POST_VIEW` to clean up actions taken by `TM_PRE_VIEW`. However, if an error is encountered during `TM_VIEW` processing, the processing immediately switches to `TM_POST_VIEW`.

Processing the Event Stack

When an error occurs (setting `TM_STATUS` to a non-zero value), transaction events on the event stack continue to be processed. Since all slices for a request can be pushed onto the stack at the same time, processing for each slice would occur even if there is an error.

If there are no events on the stack, then processing continues as if the original event had failed. You can clear the event stack by calling `sm_tm_clear_model_events`.

Controlling Error Messages

If the `TM_STATUS` variable is not zero when `sm_tm_command` completes its processing of a command, the transaction manager displays an error message. The content of the message indicates the first error that the transaction manager encountered.

This is the standard behavior for errors in the transaction manager. You can change this behavior by setting the following variables:

- `TM_EMSG_USED`—Controls whether the error message is displayed.
- `TM_MSG_TEXT`—Unconditionally sets error message text.

- `TM_PROPOSE_MSG_TEXT`—Sets error message text only if a message text has not already been set.
- `TM_MSG`—Unconditionally sets the error message number.
- `TM_PROPOSE_MSG`—Sets the error number only if a previous error number was not set.

Error Message Display

To set whether the transaction manager error message be displayed, use:

```
sm_tm_iset(TM_EMSG_USED, flag)
```

If flag is zero, then `sm_tm_command` displays an error message. If flag is non-zero, then `sm_tm_command` does not display an error message, even if an error is encountered.

The most common use of this facility is to disable the transaction manager error message in cases where the error message has already been reported to the user and the transaction manager error message is merely a duplicate.

An example of this could be part of a database error handler as illustrated in the following JPL procedures. The entry procedure is called on screen entry and activates the error handler. The `dberror` procedure specifies the content of the error handler and tests for a connection error. If a database connection does not exist, the error handler displays the database driver error, but disables the duplicate transaction manager error.

```
proc entry
   DBMS ONERROR JPL dberror
return 0
proc dberror
if @dmretcode == DM_NO_CONNECT
{
   call sm_tm_iset(TM_EMSG_USED, 1)
   msg emsg "Panther DB: " @dmretcode " " @dmretmsg "%N"\
   "Engine Error: " @dmengretcode " " @dmengretmsg}
else
   msg emsg "Panther DB Error: " @dmretcode " " \
@dmretmsg "%N"\
   "Engine Error: " @dmengretcode " " @dmengretmsg
```

Error Message Content

You can specify the text of an error message using the following:

```
sm_tm_pset(TM_MSG_TEXT, text)
```

```
sm_tm_pset(TM_PROPOSE_MSG_TEXT, text)
```

Setting `TM_MSG_TEXT` specifies the content of the message in all conditions. Setting `TM_PROPOSE_MSG_TEXT` specifies the content of the message only if a message has not already been specified.

When an error occurs in a transaction event, the transaction manager always sets `TM_PROPOSE_MSG_TEXT` which in turn sets `TM_MSG_TEXT` only if it is empty. It is the value of `TM_MSG_TEXT` that is displayed when `sm_tm_command` finishes.

The transaction models also use this call after calling `dm_dbms` to execute SQL statements. They thus preserve non-empty values that might have been set by an error handler (if some other non-empty value had not already been set in `TM_MSG_TEXT`).

To change the message displayed for a transaction manager error, you must set these error message variables. The error message variables take precedence over the error number variables.

Error Message Numbers

If no message text has been set by a call to `sm_tm_pset` *or by an error in the transaction manager itself*, error message display can be specified by calling:

```
sm_tm_iset(TM_MSG, msg_nbr)
```

```
sm_tm_iset(TM_PROPOSE_MSG, msg_nbr)
```

The `msg_nbr` parameter would correspond to a message number defined in a message file.

- `TM_MSG` sets the message number in all conditions.
- `TM_PROPOSE_MSG` sets the message number if no previous message number has been established. When `sm_tm_command` finishes processing the command, the message corresponding to the specified number is displayed.

The text of associated with error messages can be defined in a message file.

Refer to page 45-2 for details on how to create a message file for your application.

Suppress Error Messages

To suppress transaction manager messages, set:

```
sm_tm_iset
```

```
(TM_EMSG_USED, flag)
```

in a customized transaction model, in the database error handler, or in an event function. If flag is set to a non-zero value, the transaction manager does not display an error message.

CHAPTER

37 Processing Application Errors

While users of your application need to be notified when errors occur, it is also important to know what is happening on your servers, both during development and at runtime. Errors that occur remotely may have no meaning to a user, but in most cases, you want to determine which errors should be posted or broadcast from server to client, and what information should be displayed that will be most useful to a user of your application.

Panther provides global variables and hook functions to help you manage errors that result from requests made to a database. Default error handlers are installed to log errors on your server from Panther's database drivers and from the database engine. In addition, you can write (in JPL or C) and install customized error handlers. With a single JPL procedure, you can change the way errors are handled and control which messages are logged.

This chapter discusses:

- Built-in handlers for logging database errors that occur on the server; and the built-in error handler for handling database errors that occur when the client makes direct calls to a database.
- Global variables that contain error and status information from Panther's database drivers and from the database engine.
- Database hook functions and information for writing your own hook functions.
- Custom database error handlers for your application.

Default Error Handlers

The behavior of the default error handlers depends on where the database interaction occurs, that is, on which agent (client or server) and whether, particularly for servers, the application is in development or production.

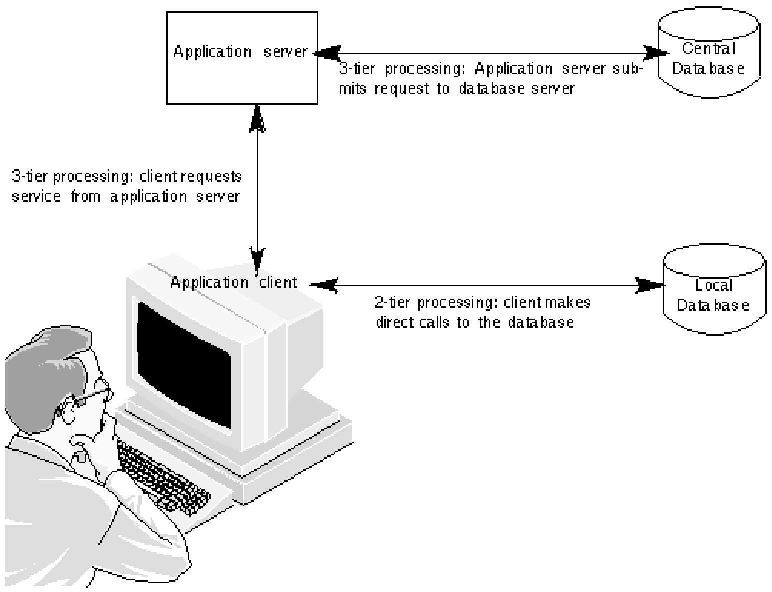

Application server
3-tier processing: Application server submits request to database server
Central Database
3-tier processing: client requests service from application server
Application client
2-tier processing: client makes direct calls to the database
Local Database

Figure 37-1 By default, database errors are logged on the server. The application client outputs database errors only if database calls were made directly from client to database.

When database requests are submitted to the database via a service request from an application server, error processing takes place on the server. Therefore, there is no direct output to the client. To ensure that your application reports these errors, error handlers are installed on the server at initialization. The handlers let you monitor the results of DBMS events on the server by logging server activity to a central event log.

Depending on the outcome, you might also want to display pertinent information on the client by including status/error information with service call results (refer to page 26-1 for information on displaying messages to a user).

If calls are made directly to a database from the client, the Panther default hook function provides output directly to the user.

Server Activity

Panther installs different default handlers for the development environment and for your production application. The handlers log messages to the central event log. One handler logs each `DBMS` command, the other logs errors if they occur as the result of `DBMS` commands.

The default development handler for the `DBMS ONENTRY` function is `sm_tp_dbms_cmd_log`, which logs each `DBMS` command. To minimize overhead and since no logging of commands is required at runtime, no handler is installed for production applications.

The default development and production error handler for the `DBMS ONERROR` function is `sm_tp_dbms_error_print_all` which logs all `DBMS` command errors to the central event log.

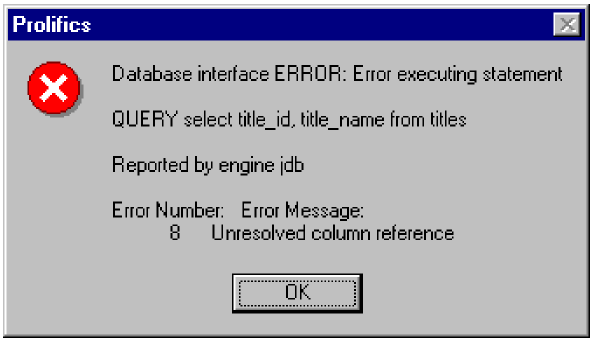
You can override the default handlers by installing your own function to handle errors. The application can also install an exit function to process all error and status information and to display this information in the application as well as log it to the central event log. Writing and installing your own handler is covered later in this chapter.

Client Output

Panther installs a default error handler which is used when executing `DBMS` commands directly from the client to a database, that is, without access to the request broker. In general, direct access to the database is not recommended in three-tier architecture since your application cannot take advantage of three-tier features, such as routing, during basic client/server processing.

If an error occurs, the default error handler displays the following information in a dialog box:

- An engine-independent error message from Panther's database driver.
- The first 255 characters of the statement which caused the error.
- The engine that reported the message, if applicable.
- The engine's error number and error message, if applicable.



Database interface ERROR: Error executing statement

QUERY select title_id, title_name from titles

Reported by engine jdb

```
Error Number:   Error Message:
        8      Unresolved column reference
```

Figure 37-2 Sample error message from the client's default error handler.

For this type of error, the default error handler returns a -1 to its caller. If an error occurs while executing JPL, Panther aborts the JPL procedure where the error occurred. An aborted JPL procedure always returns `-1` to its caller.

An application can override the default handler by installing its own function to handle errors. It can also install an exit function to process all error and status information and to display this information in the application.

Variables for Logging Error and Status Information

Panther supplies several predefined variables where it stores database error and status data for the application. These global variables (begin with the characters `@dm`) are automatically defined at initialization and maintained by Panther.

Before executing a `DBMS` statement, Panther clears the contents of all DBMS global variables. After executing a `DBMS` statement, Panther updates these variables with any error, warning or status information returned by the engine.

In addition to engine-specific codes and messages, Panther also supplies engine-independent codes and messages. These messages are defined in `dmerror.h`.

The variables and their values are available in JPL and in C via Panther library functions like `sm_n_getfield` and `sm_n_fptr`.

Table 37-1 Database global variables

Variable	Description
`@dmretcode`	Status of the last executed `DBMS` statement; value is `0` or one of the codes defined in `dmerror.h`.
`@dmretmsg`	Engine-independent message associated with last executed `DBMS` statement; value is either empty or one of the messages (defined in Panther's message file. If `@dmretcode` is `0`, this variable is empty.
`@dmengerrcode`	Engine-specific error code for last executed `DBMS` statement; value is `0` (engine did not detect any errors) or an engine-specific code.
`@dmengerrmsg`	Engine-specific error message for last executed `DBMS` statement. If `@dmengerrcode` is `0`, this variable is empty.
`@dmengwarncode`	Engine-specific warning code or bit setting for last executed `DBMS` statement. If empty, the engine did not detect any warning conditions.
`@dmengwarnmsg`	Engine-specific warning message for last executed `DBMS` statement. If `@dmengwarncode` is a byte or is blank, this variable is empty.
`@dmengreturn`	Return code from last executed stored procedure; value is either blank (engine did not supply a return code) or an integer.

Table 37-1 Database global variables

Variable	Description
`@dmrowcount`	Number of rows fetched to Panther variables by last `SELECT` or `CONTINUE` statement. It can also contain the number of rows affected by `INSERT`, `UPDATE` or `DELETE` statements.
`@dmserial`	Engine-generated value for a serial column; value is `0` or an appropriate serial value for the column.

For more information on these variables, refer to page 12-1 in the *Programming Guide*.

Database Error Event Functions

Panther provides the following database-specific error event functions which are invoked when specific database events occur:

`ONENTRY`	Called before executing any `DBMS` command from JPL or C.
`ONEXIT`	Called after executing any `DBMS` command from JPL or C.
`ONERROR`	Called if an error occurs while executing any `DBMS` command from JPL or C.

The error event functions receive three arguments:

- A copy of the first 255 characters of the command line. If the command is executed from JPL, this is the first 255 characters after the JPL command word `DBMS` or `DBMS SQL`.
- The name of the current engine. If the command used a `WITH ENGINE` or `WITH CONNECTION` clause, the argument identifies this engine. If no `WITH` clause is used, the argument identifies the default engine.

- A context flag identifying why the function was called. For an ONENTRY function, its value is 0; for an ONEXIT function, the value is 1; and for an ONERROR function, the value is 2.

ONENTRY Function

Before executing a DBMS command from JPL or C, Panther executes the application's installed ONENTRY function. An ONENTRY function is useful for logging or debugging statements. The default handler for development servers is sm_tp_dbms_cmd_log which logs each command to the central event log.

If you are supplying your own ONENTRY function, you can also use it to modify the Panther environment, for instance, to remap cursor control keys or change protection-type properties on widgets on client screens with a direct connection to a database.

ONEXIT Function

After executing a DBMS command from JPL or C, Panther executes the application’s installed ONEXIT function. An ONEXIT function is useful for logging or debugging statements. This function is also useful for checking error and status codes after each command.

If you are supplying a custom ONEXIT function, you can use it to modify the Panther environment, for instance, to remap cursor control keys or change protection-type properties on widgets on client screens with a direct connection to a database.

ONERROR Function

If an error occurs in the database driver while executing a DBMS command from JPL or C, Panther executes the application's installed ONERROR function. All errors are logged to the event log with the default handler sm_tp_dbms_error_print_all. An ONERROR function can log the values of the global DBMS error variables. It might also log the text of the command that failed.

If the error occurred while executing a command from JPL, the ONERROR function determines whether control is returned to the procedure or to the procedure's caller.

If you are using JPL, it is recommended that you install an ONERROR function. This ensures consistent error handling throughout the application and reduces the amount of code needed to handle errors. If an ONEXIT function is also installed, Panther calls the ONEXIT function before the ONERROR function.

Writing an Error Event Function

You can write database error event functions in JPL or C.

A JPL error event function is installed as follows:

```
DBMS { ONENTRY | ONEXIT | ONERROR } JPL entryPoint
```

where `entryPoint` is an entry point to a JPL module. The entry point can be a procedure name, a file name, or the name of your custom error handler. Refer to the JPL section of the *Programming Guide* for more information.

A C error event function is installed as follows:

```
DBMS { ONENTRY | ONEXIT | ONERROR } CALL function
```

where function is a prototyped function that takes three arguments: two strings and an integer. For example, the following entry in the `pfuncs` structure installs `myfunc` as a prototyped function that returns an integer:

```
static struct fnc_data pfuncs[] =
{
    ...
    SM_INTFNC("myfunc(s,s,i)",  myfunc),
    ...
};
```

For more information on installing prototyped functions, refer to page 44-9.

To turn off a custom error event function and reinstate the default handler, execute the command with no arguments. For example:

```
DBMS ONERROR
```

For more information and examples of each event function, refer to page 11-42 in the *Programming Guide*.

`ONENTRY`

The return code from an `ONENTRY` function is ignored if the current command was executed from JPL. If the command was executed from C, the return code is returned to the calling function. It is recommended that this function return 0.

`ONEXIT`

The return code from an `ONEXIT` function is ignored unless an error occurred while executing a `DBMS` command using JPL. If the return code from the function is non-zero, Panther aborts the JPL procedure where the error occurred and returns `-1` to the caller. If the command is executed from C, the return code is returned to the calling function.

If the application is also using an `ONERROR` function, the return code from the `ONERROR` function overrides the return code from the `ONEXIT` function.

`ONERROR`

If an application is using an installed error handler, the error handler determines the handling for errors that occur while using JPL.

If an error occurs in Panther's database interface while executing JPL, a non-zero return code aborts the JPL procedure where the error occurred. The procedure's caller (either Panther or another JPL procedure) gains control. If the return code is `0`, the JPL procedure resumes control; Panther executes the next statement in the JPL procedure.

If an error occurs in Panther's database interface while executing a C function, the `ONERROR` return code is returned to the calling function.

The return code from an `ONERROR` function overrides the return code from an `ONEXIT` function.

Custom Error Handlers

It is recommended that your Panther application use an error handler. If the default handlers do not accomplish what you desire, you can write a custom error handler in JPL or C. In this way you can customize the error messages appearing in your application. You can use any of the global variables as part of an error handler. For example, it can use `@dmretmsg` to display a message from Panther's database driver or `@dmengerrmsg` to display an engine-specific error message. It might also display its own messages depending on the values in `@dmretcode` and `@dmengerrcode`.

The procedure's return code determines whether or not JPL continues or aborts the procedure it was executing.

There are two classes of errors in Panther's database drivers:

- Syntax or logic error in a DBMS statement—This might include: executing a DBMS command that is not supported by the current engine, using an invalid keyword, executing a cursor that has not been declared, or failing to declare a connection before executing a DBMS statement that requires one. These errors are detected by Panther's database driver which updates the global variables @dmretcode and @dmretmsg.

- Engine error—This might include: attempting to SELECT from a non-existent table or column, inserting invalid data in a column, logging on with invalid arguments, or attempting to connect to a server that is not running. These errors are detected by the engine and passed to Panther's database driver which updates the global variables @dmretcode, @dmretmsg, @dmengerrcode, @dmengerrmsg, @dmengwarncode, and @dmengwarnmsg.

There are also Panther and JPL errors which are not a part of Panther's database driver. A JPL procedure might fail because of JPL syntax or colon preprocessing errors. If a JPL error occurs, Panther outputs an error message describing the error, the source of the JPL statement, and the statement that failed. Furthermore, it aborts the JPL procedure where such an error occurred and returns control to the procedure's caller. It is assumed that JPL and Panther errors are detected and corrected during application development. The only time that an application might need special handling for these errors is during transaction processing. For more information, refer to page 28-10.

An ONERROR function overrides Panther's default DBMS error handler. The function controls the display of error messages. If the error occurred while executing a command from JPL, the ONERROR function also determines whether control is returned to the procedure or to the procedure's caller.

If you using JPL, install an ONERROR function. This ensures consistent error handling throughout the application and reduces the amount of code needed to handle errors. If an ONEXIT function is also installed, Panther calls the ONEXIT function, then the ONERROR function.

Example

This procedure checks if the error is DM_ALREADY_ON. In this case, it logs a message and returns 0. For all other errors, it checks for an engine error code. If there is an engine error, it logs the statement and engine-specific error message.

```
proc screen_entry
   DBMS ONERROR jpl dbi_error_handler
   ...
```

```
return

proc dbi_error_handler (statement, engine)

   if (@dmretcode == DM_ALREADY_ON)
   {
      msg emsg "You are already logged on."
      return 0
   }

   if (@dmengerrcode != 0)
   {
      msg emsg @dmretmsg "%N" "Statement :statement" "%N" \
         ":engine Error :@dmengerrcode :@dmengerrmsg"
   }
   else
   {
      msg emsg "Application Error:  :@dmretmsg " \
         "See the DBA for assistance."
   }
   return 1
```

The following example illustrates how a database engine error is logged to the central event log.

```
proc dbi_error_handler(statement, engine)
{
   vars message1, message2, message3

   message1 = "DBMS ERROR " ## statement
   message2 = "Database code " ## @dmretcode ## \
      ", Database message " ## @dmretmsg
   message3 = "DBMS code " ## @dmengerrcode ## \
      ", DBMS message " ## @dmengerrmsg

   log message1
   log message2
   log message3

   // On DBMS failure, terminate service request with failure
   service_return failure ()
}
```

Part VI Testing Your Application

After building the application, you need to test it. This section gives instructions on using Test Mode in the editor and on using the debugger. It also lists some guidelines for optimizing your application.

CHAPTER

38 Testing Application Components

In the process of development, Panther allows you to simulate how your application behaves and appears to end users via test mode. From test mode, you can return to the editor, make changes to your application, and resume testing without exiting from the executable.

There are two Panther executables, `prodev` and `prorun` (`prodev32/64.exe` and `prorun32/64.exe` in Windows).

- When starting `prodev`, you are in the editor where you can begin creating your Panther application. To test an application component, bring focus to it and choose File→Test Mode (or the Test Mode button on the toolbar). The status bar displays Test Mode.
- When starting `prorun`, you are in application mode. Access to the editor is not available.

Test Mode Menu Bar

Once test mode or application mode is entered, the menu displays the default application toolbar. If your application has its own menu bar/toolbar, the Panther menu is not displayed.

The following menu options are available while in test mode:

Edit menu

Provides basic editing commands (not available in character-mode platforms) that you can use while you are testing your applications.

Options menu

Includes the following options:

- Top Screen—Accesses the top-level screen of your application, or the initial screen displayed when you entered test mode. This option clears all other screens from the form and window stacks.
- System Commands—Available only in character mode platforms. Allows you to execute operating system commands from within Panther. Enter the command, and choose OK to execute it. When the command terminates, press any key to return to test mode.
- Open Screen —Invokes the dialog box where you can choose to open any screen—as a form, a window, or sibling window. This allows you to jump anywhere within your application.
- Editor —Invokes the editor.
- Debugger —Invokes the Panther debugger. For instructions, refer to Chapter 39, “Using the Debugger.”

Keys menu

Provides access to Panther's logical keys and their functions. These are useful if your application screen does not provide other methods for carrying out commands, such as Exit, Transmit, Switch scope, or Zoom keys.

Windows menu

Allows you to bring focus to any screens that are currently open.

Transaction menu

Provides access to transaction manager commands that you can use with screens that are derived from your database. The commands are:

- View—Select one or more records for display purposes only.
- Select—Select one or more records for possible update.
- Continue—Fetch the next group of selected records (in two-tier applications only).
- New—Close the current transaction (if any), and clear existing data, to allow a user to enter information to create a new record.
- Copy—Close the current transaction (if any), without clearing existing data, to allow a user to create a new record from the existing data.
- Save—Update the database to reflect the changes or additions made by a user.
- Close—Terminate the transaction in progress
- Clear—Clear transaction data (that is, data entered in a field) from the screen. This applies only when adding a new record or when entering criteria for selecting records.

Refer to Chapter 34, “Specifying Transaction Manager Commands,” for information on writing transaction manager commands.

Database menu

Allows you to connect to, as well as disconnect from, databases. The Trace On/Off command allows you to toggle the display of SQL statements that are automatically generated by way of the transaction manager commands.

Refer to Chapter 33, “Using Automated SQL Generation,” for information on setting properties that take advantage of Panther's automatic SQL generation.

Middleware Session menu

(JetNet/Oracle Tuxedo only) Allows you to open, as well as close, a middleware session.

Report menu

Allows you to run, view or print out a report previously saved in a metafile (refer to "Running Reports from the Report Menu" on page 9-1 in *Reports*).

Testing Application Components

You can test your client screens and service components through a direct or remote connection to the database. You need to be in test mode to test how your application components and their contents function.

Notes: If the screen you are testing references other screens, the referenced screens will not be viewable unless you saved them to a library before entering test mode. Also, if you go into test mode without saving changes to the referenced screens, you will only see the last saved version of the referenced screen and not the unsaved changes.

When you want to test a screen, you bring focus to it and choose File→Test Mode (or the Test Mode button on the toolbar). If you have other open screens in your Workspace with unsaved changes, you are prompted:

```
Do you want to save changes to <screenName>@libraryName?
```

Choose the desired action:

- Yes—Saves the screen with its given name to the appropriate library.
- Yes to all—Saves all open screens (except for the one you want to test and untitled screens).
- No—Does not save the changes, but the changes are not lost. You can save the changes when you return from Test mode.
- No to all—Does not save the changes of any of the open screens, but the changes are not lost. You can save the screens when you return from Test mode.
- Cancel—Cancels the test request.

The active screen opens in Test mode.

Notes: Test mode is virtually identical in appearance and function to Application mode, except that when you exit Test mode you return to the screen editor.

Testing Screens and Service Components

In a two-tier architecture, you need a direct connection to the database to test your screens; however, in a three-tier architecture, you need a remote connection to the database to completely test your client screens.

Before you test your three-tier client screens, you must save the corresponding service components and selection service components (if any) to their appropriate libraries. For JetNet and Oracle Tuxedo applications, you must also define the services in the JIF.

You can test client screens and service components via a direct or remote connection to the database. You can test your service components with a direct database connection to check whether the appropriate data is passed from the database. A remote database connection is established by configuring and running servers which access the database and provide services.

Service Components

You can test a service component to see if the appropriate data is passed from the database to the service component. This requires a direct database connection rather than connecting to the database by way of the application server.

How to Test a Service Component with a Direct Database Connection

1. Bring focus to the screen you want to test by clicking on it.
2. Choose File⟶Test Mode (or the Test Mode button on the toolbar).

   The active service component opens in Test mode. Any changes (saved or not) made to the service component are reflected in Test mode.
3. Choose Database⟶Connect to connect to the database.

While in test mode you can:

- Test database transactions via the Transaction menu option.
- If the Panther debugger is linked into your application or authoring executable, invoke the debugger window by pressing the `DBUG` logical key. (Refer to Chapter 39, "Using the Debugger," for more information.)

4. Choose Options→Editor or press the Panther Exit key twice to exit Test mode. You return to the editor, with the workspace restored to the way it was when you exited to Test mode.

Save the service component to the server library and then you can test it with client screens.

Three-tier Client Screens

The main purpose of testing a client screen with a remote database connection is to see whether service requests are carried out and the appropriate data is returned to the client. However, to test a client screen in this manner, you must do the following as prerequisites:

■ Test and save the corresponding service component to the server library.

Notes: It is recommended that you adopt a naming convention that identifies client screens with their corresponding service components or vice-versa.

■ Update the JIF as follows (JetNet/Oracle Tuxedo only):

- Ensure that the service name specified in the JIF is the same as the service name specified in the Service property for table view widgets and link widgets on the client screen and selection screens (if any).
- Check to see if the procedure name is accurate.
- Check to see if the name of the service component is accurate.
- Ensure that the service routine specified in the Service property for table view widgets on the client screen and selection screens (if any) matches the transaction type specified in the JIF.

Now, test the client screen.

How to Test Client Screens with a Remote Database Connection

1. Bring focus to the screen.
2. Choose File⟶Test Mode (or the Test Mode button on the toolbar).

   The active screen opens in Test mode.
3. Choose Middleware Session⟶Connect to connect to the middleware if you are not already connected.

   While in test mode you can:

   - Try out the screen to see if it operates as planned. Check the tabbing order, data entry formats, control strings, expressions, etc.
   - Invoke screens attached to the current screen; however, unsaved changes made to other screens are not reflected in Test mode.
   - Test a menu bar and toolbar that you attached to this screen.
   - Test database transactions via the Transaction menu option, Commands menu option (if you specified a menu bar), or push buttons.
   - If the Panther debugger is linked into your application or authoring executable, invoke the debugger window by pressing the `DBUG` logical key. (Refer to Chapter 39, “Using the Debugger,” for more information.)
4. Choose Options⟶Screen Editor or press the Panther Exit key twice to exit Test mode. You return to the screen editor, with the workspace restored to the way it was when you exited to Test mode.

Two-tier Client Screens

The main purpose of testing a client screen with a direct database connection is to test database transactions via the Transaction menu. You check to see if the appropriate data is passed from the database to the client screen.

How to Test Client Screens with a Direct Database Connection

1. Bring focus to the screen by clicking on it.
2. Choose File⟶Test Mode (or the Test Mode button on the toolbar).

The active screen opens in Test mode.

3. Choose Database⟶Connect to directly connect to the database if you are not already connected.

   While in test mode you can:

   - Try out the screen to see if it operates as planned. Check the tabbing order, data entry formats, control strings, expressions, etc.
   - Invoke screens attached to the current screen; however, unsaved changes made to other screens are not reflected in Test mode.
   - Test a menu bar and toolbar that you attached to this screen.
   - Test database transactions via the Transaction menu option, Commands menu option (if you specified a menu bar), or push buttons.
   - If the Panther debugger is linked into your application or authoring executable, invoke the debugger window by pressing the `DBUG` logical key. (Refer to Chapter 39, "Using the Debugger," for more information.)

4. Choose Options⟶Editor or press the Panther Exit key twice to exit Test mode. You return to the editor, with the workspace restored to the way it was when you exited to Test mode.

Closing and Exiting

How to Close a Screen, but Remain in the Editor

1. Bring focus to the screen by clicking on it.
2. Choose File⟶Close⟶Screen.

   If you made changes to the screen, and have not saved them, the following message appears:

   ```
   Do you want to save changes to <screenName>?
   ```

   - Choose Yes to save the screen. If the screen is new, the Save As dialog box opens where you can name the screen in a library.
   - Choose No to close the screen without saving your changes.
   - Choose Cancel to keep the screen open.

How to Exit from the Editor

1. Choose File→Exit.

If you have changed any open screens and have not saved them, Panther prompts you to save them now.

How to Exit Application Mode

Choose Close or Quit from the system menu.

CHAPTER

39 Using the Debugger

Panther's debugger provides you with the ability to view and analyze all application components. You can view the application's execution from various perspectives with varying degrees of detail. You can examine and debug client and server components. You can:

- Step through events, stopping to examine data.
- Set breakpoints at events or JPL code on which to interrupt program execution. Breakpoints can be set on screen and widget events, program execution events such as database and transaction manager events, as well as source code locations. Breakpoints on screen, widget, group, and grid events can be further restricted to specific subevents.

  Note: Events in the JetNet middleware API are not recognized in the debugger.
- Examine application data in Panther variables and arrays, screen and service component properties, widget properties, and JPL variables or expressions
- Step through your JPL code. You can view source code modules from open libraries, memory resident modules, public JPL modules, the window stack, and the program stack—as well as set breakpoints on their contents.
- Call installed functions and JPL procedures and evaluate a JPL expressions.
- Stop at a change in a variable or expression.
- Review debugger activity in the log file. Debugger messages can be written to a user-specified log file.

This chapter describes the features of the Panther debugger and how to use it. If you are developing a two-tier application, the debugger is already linked in with your development executable (`prodev`) so you can debug client screens. If you are developing a JetNet or a Oracle Tuxedo application, you need to run a standard server in debug mode to debug service components and services.

Debugging Services and Service Components

If you are developing a JetNet or Oracle Tuxedo application, you can run a debuggable server so that you can monitor what the server is doing while it is processing service requests. Running the debugger on a server lets you see the service components that are on the server machine.

You can run the debugger on a server provided the following two conditions are true:

- A standard server is initialized in debug mode. For information on initializing a debuggable server in JetNet, refer to "Server Details" on page 3-22 in *JetNet/Oracle Tuxedo Guide*; if you are using Oracle Tuxedo, refer to "Initializing Servers" on page 8-17 in *JetNet/Oracle Tuxedo Guide*.
- The server environment file sets the `DISPLAY` variable to tell the X server where to display debuggable service components. For example:

  ```
  DISPLAY=mimosa:0.0
  ```

When running a debuggable server, your client might timeout waiting for a response from the server while you are in debugging mode.

To ensure that the client does not timeout while debugging the server:

Set the appropriate timeout options in your client code. For example:

- For `xa_begin`, set the `TIMEOUT` parameter to specify a large or infinite timeout.
- For `service_call`, use the `NOTIMEOUT` option to prevent blocking timeouts.

- For wait, set the `TIMEOUT` parameter to specify a large or infinite timeout.

The application's default blocking timeout is established for the application as part of the server configuration. For information on setting the Default Blocking Timeout parameter, refer to “Default Blocking Timeout” on page 3-12 in *JetNet/Oracle Tuxedo Guide*. For Oracle Tuxedo users, refer to your Oracle Tuxedo documentation for information on `SCANUNIT` and `BLOCKTIME` configuration parameters.

How the Debugger Works

Many debuggers do their work by invoking the applications they analyze. The Panther debugger is invoked by your application, that is, by Panther itself. Panther notifies the debugger of each significant event in the application, as it occurs. Each time it is called in this way, the debugger analyzes the event and saves any information it needs. In most cases, the debugger then immediately returns control to the application. Some events, however, cause it to emerge or awaken from the background to halt the application and display its state.

Panther is shipped with the debugger already linked in for client executables. To enable debugging for server executables, your standard server (refer to “Server Details” on page 3-22 in *JetNet/Oracle Tuxedo Guide*) must be initialized as a debuggable server.

Starting and Stopping the Debugger

The debugger runs in the background during test and application modes, you can direct the debugger to run automatically, or you can manually break execution. The debugger follows the directions you give it and awakens accordingly—for instance, if an event triggers a breakpoint you have defined, or if the value of an expression you are monitoring has changed. The debugger comes to the foreground and awaits your action: provide additional analysis of the application, modify data, set breakpoints, or put it back to sleep and return control to your application.

To enable the debugger from the screen editor:

Choose Options⟶Enable Debugger. This instructs the debugger to awaken immediately upon entry into test mode.

To enable and access the debugger in test or application mode:

Choose Options⟶Debugger, or press the `DBUG` key. To put the debugger to sleep and return to test mode, choose File⟶Resume Application.

If your application's menu bar is displayed, you can enter the debugger by pressing the `DBUG` key or switch menu bar scope by pressing the `SFTS` logical key, from where you will have access to Options⟶Debugger.

To end a debugger session

Do either of the following:

- Choose File⟶Resume Application to put the debugger to sleep and continue running/testing your application. Panther preserves the debugger state; if the debugger is reentered, all breakpoints and configuration preferences remain unchanged. Moreover, all breakpoints and break events are ignored until you explicitly reactivated the debugger.
- Choose File⟶Exit Application to about the current JPL execution and quit the debugger. If you entered the debugger from application mode, you can choose to resume in application mode or exit Panther to the operating system. If you were in test mode, you return to the screen editor. Use this option to break out of infinite loops.

Views into Your Application

The debugger provides several windows to monitor application execution, each offering a different view into the application. The windows can be opened and closed from the debugger's View menu by choosing from these options:

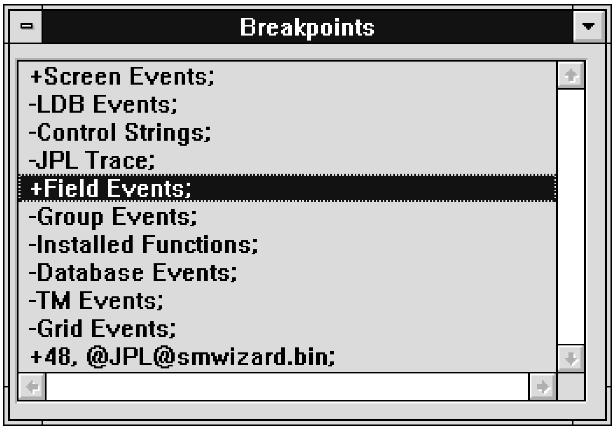
Status
: Displays the current debugger operation or event being traced.

Source Code
: Displays screen- or widget-level JPL, library or external module JPL in the Source Code window. You can set or unset a breakpoint at the current line in

the displayed code (at the cursor position) by choosing Breaks⟶Toggle Location Break or by double clicking anywhere in the line. Breakpoints are identified by an asterisk following the line number. For more information on using the Source Code window, refer to page 39-11, “Viewing JPL.”

Breakpoints

Lists all breakpoints. In normal mode, predefined breakpoints are listed: screen events, LDB events, control strings, JPL trace, field events, group events, installed functions, database events, TM events and grid events. Location breakpoints that are set as well as breakpoints you create in expert mode are also listed. In expert mode, the predefined list of breakpoints are not displayed. A plus sign (+) preceding a breakpoint indicates active, and a minus sign indicates inactive.



```
+Screen Events;
-LDB Events;
-Control Strings;
-JPL Trace;
+Field Events;
-Group Events;
-Installed Functions;
-Database Events;
-TM Events;
-Grid Events;
+48, @JPL@smwizard.bin;
```

Figure 39-1 The Breakpoints window shows currently identified breakpoints.

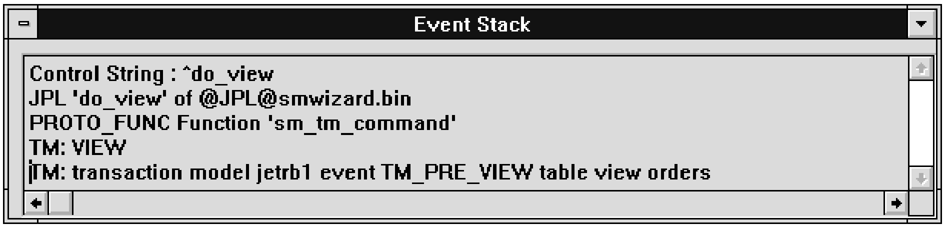
Activate or deactivate a selected breakpoint by choosing Breaks⟶Enable or Breaks⟶Disable or by double clicking on the item. Use Breaks⟶Select All if you want to enable or disable all listed breakpoints.

Choose Breaks⟶Show Source to view the source code associated with a location breakpoint.

For more information on setting breakpoints, refer to page 39-20, “Setting Breakpoints.”

Data Watch
: Monitor the values of any variables, JPL expressions, or values of properties. Enter the name of the variable or the expression. The values are updated and displayed as execution proceeds.

Event Stack
: Displays the hierarchy of nested calls to procedures and/or control strings during JPL or control string execution.



```
Control String : ^do_view
JPL 'do_view' of @JPL@smwizard.bin
PROTO_FUNC Function 'sm_tm_command'
TM: VIEW
TM: transaction model jetrb1 event TM_PRE_VIEW table view orders
```

Figure 39-2 View the call hierarchy in the Event Stack window.

Pending Keys
: Displays which keys are pushed onto the input queue.

Configuring the Debugger

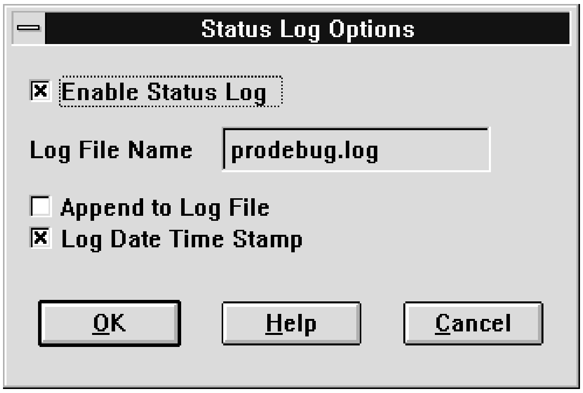
You set debugger operation preferences via the Options menu, and they are saved in the file `prodebug.cfg` in the current working directory. You should not manually modify this file.

Setting Log File Preferences

Debugger activity can be logged to a user-specified file.

To access the log file preferences:

Choose Options→Status Log. The Status Log Options dialog box opens.



[✓] Enable Status Log

Log File Name: prodebug.log

[ ] Append to Log File

[✓] Log Date Time Stamp

Figure 39-3 The Status Log Options dialog allows you to specify your log file preferences.

From the Status Log options dialog, you can:

- Enable the log file—from the start of the session, all events are logged.
- Specify the log file to which debugger activity messages should be logged. The default file is prodebug.log in the current working directory.
- Specify whether or not to append to the existing log file or overwrite it.
- Specify whether to log a date/time stamp to each new entry in the log file. Setting a date/timestamps can affect performance on some platforms.

To view the contents of the log file:

Choose File→Open→Log File anytime while using the debugger.

Setting Debugger Preferences

The following debugger preferences are available from the Options menu:

Save Preferences on Exit

Saves window configurations and other debug settings that are in effect when you exit this session. This option also saves the contents of any Data Watch and Breakpoints settings.

Expert Mode
: Runs the debugger in expert mode.

Auto Raise/Close
: Provides automatic window management. When set, the debugger determines which windows contain any relevant data at the moment, and raises them accordingly. It also closes those that were open during the last debugger invocation if they do not have any relevant data.

  When this option is disabled, the debugger, when awakened, restores its windows exactly as you left them.

Animation
: Executes the debugger in animation mode. The debugger shows changes to its windows as they occur, while waiting for a breakpoint to be executed.

Trace Database Warnings
: Includes all warnings in database events tracked by the debugger.

Trace TM Warnings
: Includes all warnings in transaction manager events tracked by the debugger.

Debugger Menu Bar

The debugger menu bar provides easy access to debugging operations. Following is a brief overview of the File, View and Tools menu options. The features controlled by the remaining menu options are described in their own task-defined sections; a cross-reference is defined for these options.

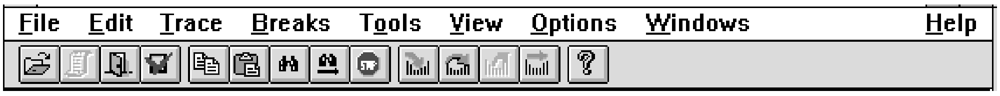


Figure 39-4 The debugger menu bar includes a Tools menu option when the debugger is run in expert mode.

File

The operations associated with the File menu are:

Open

Allows you to open a source module, the current source code, or the log file. Information is displayed in the Source Code window. For more information on Source Code window operations, refer to page 39-11, “Viewing JPL.”

Close Window

Closes the active window. If you attempt to close the last remaining window, the debugger prompts for a confirmation that you want to quit the debugger.

Save Preferences

Saves window configuration and other debug settings that are in effect at the present time for future debugger sessions. With this option, you can establish overall preferences, but make changes in the present session that will not affect those saved. Contrast with Options⟶Save Preferences on Exit, which saves them at the state they are in when you exit. For more information on setting debugger configurations, refer to page 39-6, “Configuring the Debugger.”

Resume Application

Puts the debugger to sleep and lets you continue executing your application.

Exit Application

Exits the debugger; terminates execution of your application.

Tools

The Tools menu is only available when running the debugger in expert mode, and includes the following features:

Application Data

View and access any variable displayed in the Source Code window. For information on using the Application Data window, refer to page 39-27, “Modifying and Monitoring Application Data.”

Call

Invokes the Call Installed Function window where you can enter the name of any available prototyped function along with any arguments. The return value, if any, is displayed in the status window after the function returns. The function call is logged to the log file if it has been enabled.

Sort Breakpoints
: Sorts all breakpoints listed in the Breakpoints window. Activated breakpoints are listed alphabetically, followed by inactive breakpoints.

Sort Watch Data
: Sorts all variables and/or expressions listed in the Data Watch window. For further information on this window, refer to page 39-26 .

Write Windows to Log
: Writes the contents of debugger windows to the log file. The log file must be enabled; to enable the log file, choose Options⟶Status Log Options and set Enable Status Log on the Status Log Options dialog.

View

The View menu provides several windows to monitor your application's execution. Refer to page 39-4 for a full description of the options available from the View menu.

Windows

Options available from the Windows menu allow you to:

- Bring a chosen window into focus.
- Arrange debugger windows.
- Choose an application window to examine its programmatic components. For information on viewing application screen information, refer to page 39-16.

Edit

The Edit menu commands provide standard file editing operations, such as Copy, Paste, and string search commands Find and Find Next, that you can use when accessing text data in the Source Code window. For more information on the Source Code window, refer to page 39-11.

Trace

The Trace options allow you to step through program execution one event or breakpoint at a time. For more information on tracing, refer to page 39-19.

Breaks

Breaks menu commands let you establish and manipulate breakpoints in your application—places where execution will be interrupted and the debugger will assume control. For further information on setting breakpoints, refer to page 39-20.

Options

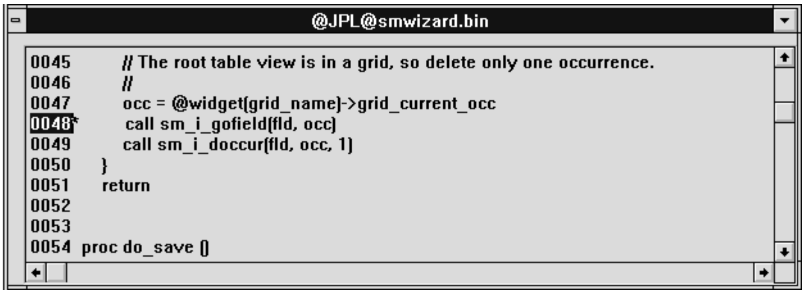
Provides options for setting debugger preferences. For information on debugger preferences and configuration, refer to page 39-6 .

Viewing JPL

To access JPL in libraries, the module must be in binary format and must also include the JPL source code. For information on compiling JPL source code, refer to page A-20.

To view screen- and widget-level JPL validation, or library JPL modules:

Choose View⟶Source Code. The Source Code Window opens.



@JPL@smwizard.bin

```
0045        // The root table view is in a grid, so delete only one occurrence.
0046        //
0047        occ = @widget(grid_name)->grid_current_occ
0048*       call sm_i_gofield(fld, occ)
0049        call sm_i_doccur(fld, occ, 1)
0050     }
0051     return
0052
0053
0054  proc do_save ()
```

Figure 39-5 The Source Code window displays a JPL module with a breakpoint set at line 48.

The Source Code window can display any JPL code contained within your application. It can be:

- Current source code — Source currently being executed. Current source code is automatically displayed when the debugger is stepping through it.
- Active source code — Code that is on the program stack, where it might be waiting for current or intermediate JPL or some other event to complete.
- Inactive source code — JPL procedures that are part of the application but have not yet been called or have already returned. If inactive source code is read into the Source Code window, it is referred to as called-up source code. To read called-up source code into the Source Code window, choose File⟶Open Source Module. The debugger’s file browsing mechanism is displayed. In this way you can also view any text files in the Source Code Window (including C source code files), but you cannot set breakpoints in them.

You can set breakpoints on any line of code that is displayed in the window. The Edit menu provides string search capabilities.

Opening a Source Module

From the Open Source Module dialog box you can specify a module to read into the Source Code window. You can also select a module from which to establish a location breakpoint.

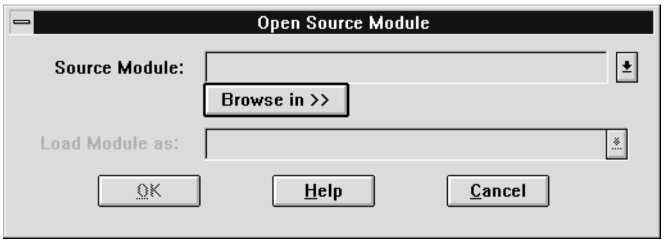


Figure 39-6 Specify a module to read into the Source Code window or select a module for the Edit Breakpoint window in Location mode.

To read a JPL module into the Source Code window:

1. Choose File→Open→Source Module. The Open Source Module Window opens.
2. Choose the Browse In button and select the location of the module:
   - Open Libraries — Select the JPL module or screen from among any open library.

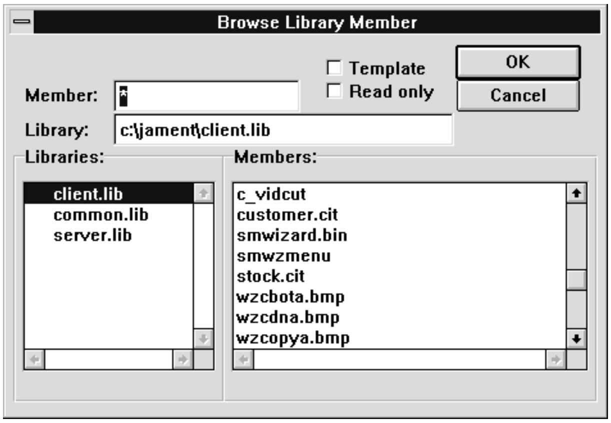


Figure 39-7 The Browse Library Member window shows open libraries and their members.

   - Memory Resident Modules—Screens compiled into your application.
   - Public Files—JPL files loaded by the public command.
   - Program Stack—Currently active JPL.
   - Window Stack—Open screens.

A Browser dialog opens. The title bar on the browser window reflects the location chosen. If no source code is available for the chosen location, the debugger informs you.

JPL modules stored in libraries must include the source as well as the binary format. By default, JPL modules created within the screen editor are stored with both formats. However, if compiled JPL is put in a library outside of the editor (using the `jpl2bin` and `formlib` utilities), do not use the `-r` flag with `jpl2bin`.

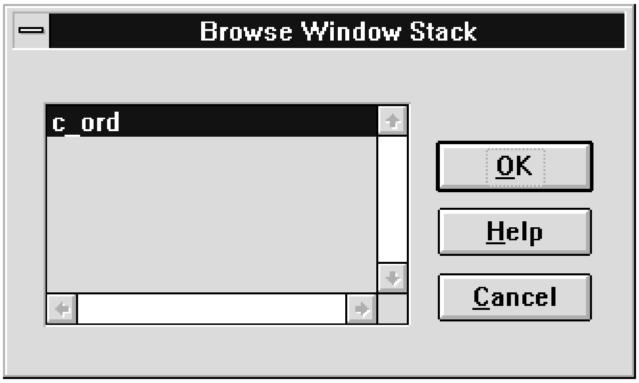
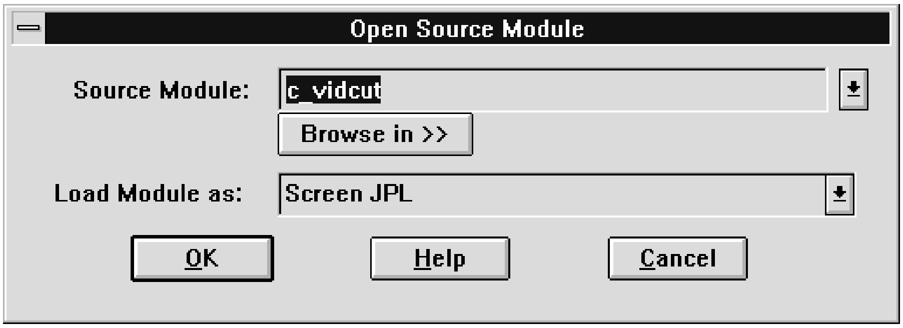


Figure 39-8 The Browse window displays the available JPL code corresponding to the browser location selection.

3. Select the JPL module from the library or the browser and choose OK. The type of the module selected is displayed set in the Load Module As option menu, and indicates one of the following types: JPL file, screen JPL, field (widget) JPL, grid JPL, or text file.

   If a request to open a screen contains JPL modules of more than one type, for instance screen- and widget-level JPL, the choices are listed in the Load Module As option menu. Select the desired type of JPL module.

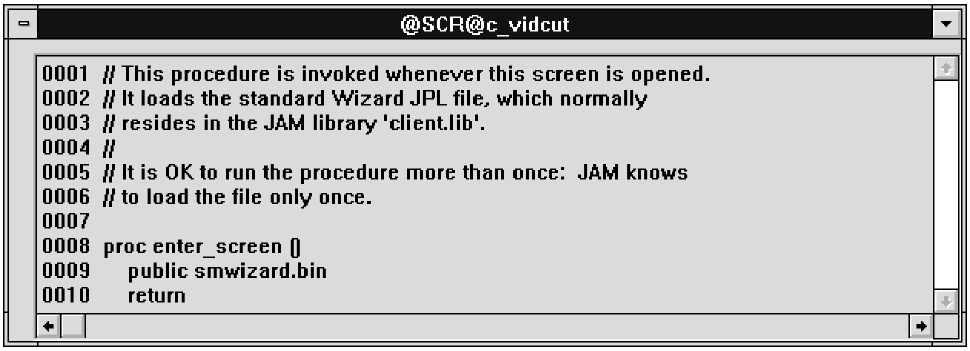


Source Module: c_vidcut

Load Module as: Screen JPL

Figure 39-9 Screen JPL associated with a screen or service component can be selected for viewing.

4. Choose OK. The Source Code window displays the contents of the specified module.



```
// This procedure is invoked whenever this screen is opened.
// It loads the standard Wizard JPL file, which normally
// resides in the JAM library 'client.lib'.
//
// It is OK to run the procedure more than once:  JAM knows
// to load the file only once.

proc enter_screen ()
    public smwizard.bin
    return
```

Figure 39-10 The Source Code window displays screen JPL code from a specific screen.

on page 39-24To set breakpoints in the Source Code window:

Choose Breaks⟶Toggle Location Break, or double-click anywhere in the line. This option toggles a breakpoint on or off on the current line of code. Breakpoints are identified by an asterisk following the line number.

Viewing Application Screen Information

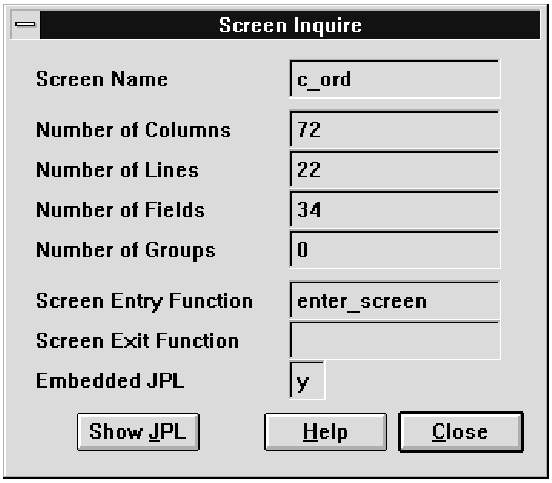
The debugger provides access to the programmatic components of your application, such as screens and service components. To view the programmatic details, choose Windows→Application and select the desired item from the list. The selected item comes to the foreground and you can access details for the following options from the Application Window menu:

Screen JPL

Displays JPL in the Source Code window. This is an alternative to using File→Open→Source Module (refer to page 39-12).

Screen Information

Displays screen information in the Screen Inquire window. If the screen has its own JPL, the Show JPL button is enabled.

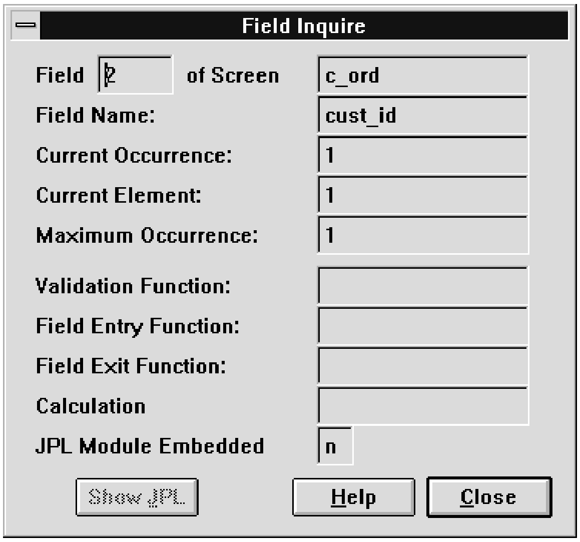


Screen Name	c_ord
Number of Columns	72
Number of Lines	22
Number of Fields	34
Number of Groups	0
Screen Entry Function	enter_screen
Screen Exit Function	
Embedded JPL	y

Figure 39-11 The Screen Inquire window displays useful information about the selected screen.

Field Information

Displays information related to widgets on your screen in the Field Inquire window. If the widget has its own JPL, the Show JPL button is enabled.

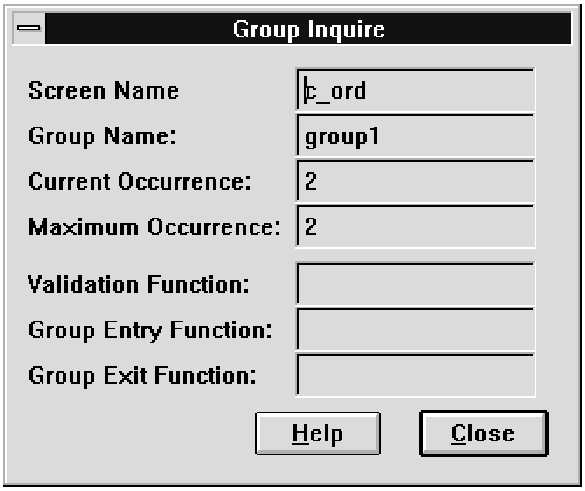


Field 2 of Screen	c_ord
Field Name:	cust_id
Current Occurrence:	1
Current Element:	1
Maximum Occurrence:	1
Validation Function:	
Field Entry Function:	
Field Exit Function:	
Calculation	
JPL Module Embedded	n

Figure 39-12 The Field Inquire window displays useful information about widgets on a screen.

Group Information

Displays information related to groups on the screen in the Group Inquire window.

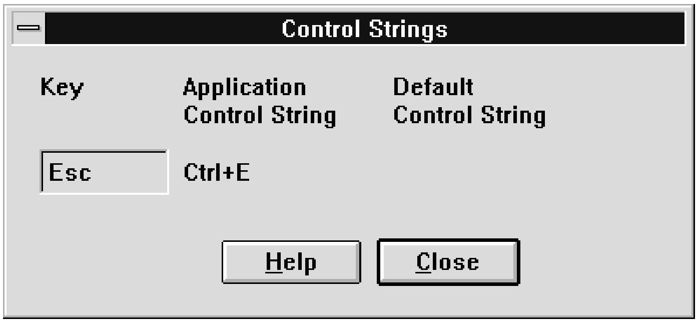


Screen Name	c_ord
Group Name:	group1
Current Occurrence:	2
Maximum Occurrence:	2
Validation Function:	
Group Entry Function:	
Group Exit Function:	

Figure 39-13 The Group Information window.

Control Strings

Invokes the Control Strings window, which shows all function keys and Panther logical keys that are assigned control strings at the application- and screen-levels.



Key	Application Control String	Default Control String
Esc	Ctrl+E	

Figure 39-14 The Control Strings window shows that Ctrl+E is mapped to the Esc key.

Done

Returns to the debugger menu bar.

Notes: To see additional properties or attributes of any screen or widget you can use the JPL properties syntax to define an expression in the Data Watch or Application Data windows.

Refer to page 39-26 for information on using the Application Data window.

Refer to page 39-27 for information on using the Data Watch window. For information on JPL properties syntax, refer to page 19-33.

You can use Data Watch to inspect a property of the application, a screen, or widget, with an expression using the JPL properties syntax.

Stepping through Program Execution

When the debugger breaks program execution and comes to the foreground, you can step through the execution of your application with the options provided by Trace menu (or via the debugger toolbar):

To Any Event
: If the current context is a JPL procedure or control string, the debugger steps to the next executable statement or string and stops. Otherwise, the debugger stops at the next event.

To Breakpoint
: Program execution resumes until the next breakpoint is reached.

If you run the debugger in expert mode you can fine tune tracing to differentiate between parent, child, and same-level events. Choose the appropriate level:

To Event→Any Level

If the current context is a JPL procedure or control string, the debugger steps to the next executable statement or string and stops. Otherwise, the debugger stops at the next event. This option is the same as Trace→To Any Event in normal mode.

To Event→This Level

If the current context is a JPL procedure call, the debugger “steps over” that procedure's statements, and breaks on the next statement at the same level, if any. If the context is a control string, the debugger steps over any strings embedded within it. Otherwise, the debugger breaks at the next event ignoring sub-events.

To Event→Higher Level

Breaks only at the next level up in the event stack. If execution is already at the topmost level, program execution resumes, breaking only for JPL breakpoints.

Using Animation

You can set the debugger to run on automatic pilot, where application execution events can be observed hands-off in the various View windows.

To enable automatic stepping:

1. Choose Options→Animation.
2. Choose Trace→To Breakpoint. The debugger refreshes all open windows on each execution event until the next breakpoint occurs.

   For example, if the Source Code window is open, the debugger scrolls through each line in the current JPL procedure as it executes; if the Data Watch window is open, variable values are refreshed whenever they change.

Animation proceeds until you toggled it off from the Options menu or interrupt it with the `EXIT` key.

Setting Breakpoints

The debugger recognizes all major execution events in a Panther application as potential breakpoints, with the exception of request broker events.

Breakpoints can be set at:

- Locations in code: JPL statements (screen- and widget-level as well as in library modules).
- Program execution events and subevents.
- Value changes in a variable or expression.

The debugger gets control at every execution event and potential breakpoint. It then decides whether or not to awaken, and whether or not to break execution. The decision is based on your Trace choice, and whether breakpoint conditions are satisfied.

If you are not running the debugger in normal mode, you can set location breakpoints in the Source Code window and modify the predefined execution events in the Breakpoints window. In expert mode, you can you can add and modify breakpoints of both types via the Edit Breakpoint window.

Setting Location Breakpoints

Location breakpoints are JPL statements that are marked in order to stop program execution. The debugger comes to the foreground whenever Panther encounters them. You can set breakpoints on any JPL code that is displayed in the Source Code window—on both called-up source code as well as current (active) source code.

Refer to page 39-11 for information on viewing JPL.

To set or clear a location breakpoint:

Select a line of code in the Source Code window and choose Breaks→Toggle Location Break, or double click on the line. The breakpoint is toggled on or off. Breakpoints are indicated with an asterisk (*).

Setting Breakpoints on Execution Events

You can set breakpoints at a variety of execution events. A finer control of event filtering is available for certain GUI events when running the debugger in expert mode. Request broker events are not recognized by the debugger.

Table 39-1 list the types of execution events on which you can set breakpoints

Table 39-1 Predefined execute events on which to set breakpoints

Event type	Description
Screen events	Breaks on all screen entry and exit events. Screen events have subevents defined, which you can selectively enable in expert mode.
LDB events	Breaks on each LDB read or write operation.
Control strings	Breaks on execution of each control string. If you have em bedded control strings, the debugger breaks on each depending on the current step level.
JPL trace	Breaks on every line of JPL execution. To step through each line of code, choose Trace⟶Breakpoint. To break at a specific line, toggle break on the line in the Source Code window and turn off JPL Trace in the Breakpoints window. Trace to Breakpoint continues execution and breaks at the specific line of code.
Field events	Breaks on all widget entry, validation, and exit events. Field events have subevents defined, which can be accessed in expert mode.
Group events	Breaks on all group entry and exit events. Group events have subevents defined.
Installed functions	Breaks when Panther is about to call any C function (installed by calling `sm_install` or identified in fun `clist.c`). The debugger recognizes calls to automatic functions, for example, the automatic screen function that is called on all screen entry and exit events, as well as other contexts where a C function can be called, such as in a control string with a JPL call statement, or upon field entry.
Database events	Breaks on any database interface event, including DBMS commands and SQL statements.
TM events	Breaks on any transaction manager event: transaction manager commands, requests, and slices.
Grid events	Breaks on any grid widget events. Grid events have subevents defined, which can be accessed in expert mode.

To set a breakpoint on a specific event type:

1. Choose View⟶Breakpoints. The Breakpoints window is displayed with the predefined events listed.

2. Select the event from the list. Do either of the following:

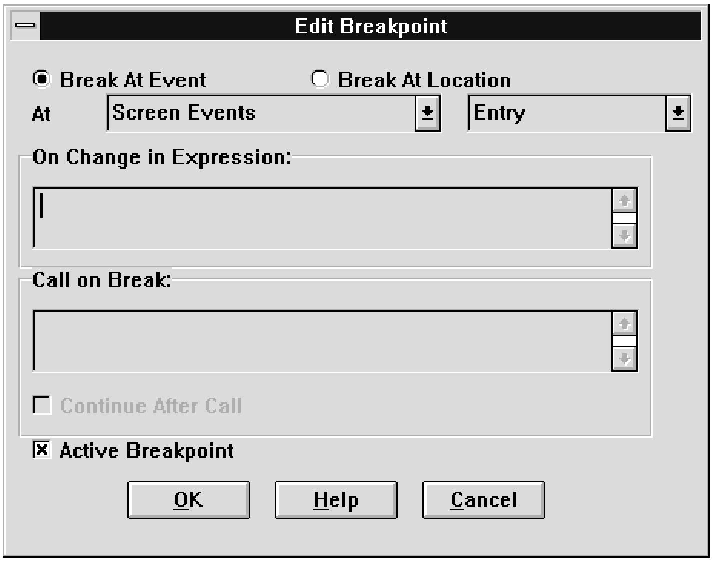
   - Choose Breaks→Enable.
   - Double-click on the item.

   Disable a breakpoint by double-clicking on it or choose Breaks→Disable. An activated event breakpoint is indicated by a plus sign (+) prefix; an inactivate breakpoint is prefixed with a minus sign (-).

To add and modify breakpoints (in export mode):

1. Choose Options→Expert Mode. Choose Breaks→Add or Edit. The `Edit Breakpoint` dialog opens.

You can add and edit both location and event breakpoints, and perform other actions. There are two modes: Event and Location. Select the appropriate check box: Break At Event for events (the default setting), and Break At Location to edit source code breakpoints.



(● Break At Event) ○ Break At Location

At: Screen Events — Entry

On Change in Expression:

Call on Break:

☐ Continue After Call

☒ Active Breakpoint

Figure 39-15 In expert mode, use the Edit Breakpoints dialog to add or modify breakpoints of all types.

To break on specific events or subevents (in expert mode):

1. From the Edit Breakpoints dialog box, choose the Break at Event radio button.

2. Select the event from the At option menu. If the selected event has subevents defined, a subevent option menu is available. The events and their subevents are listed in Table 39-2.

Table 39-2 Events and corresponding subevents available for breakpoints

Screen events	Field events	Group events	Grid events
Any subevent	Any subevent	Any subevent	Any subevent
Entry	Entry	Entry	Grid entry
Exit	Exit	Exit	Grid exit
	Validation	Validation	Row entry
	Calculation		Row exit
			Validation

3. (Optional) Indicate by checking or unchecking the Active Breakpoint check box if the item should be added to the list of breakpoints in an active or inactive state.

4. Choose OK. The event is added to the list of breakpoints (in the Breakpoints window).

To set a breakpoint at a specific location (in expert mode):

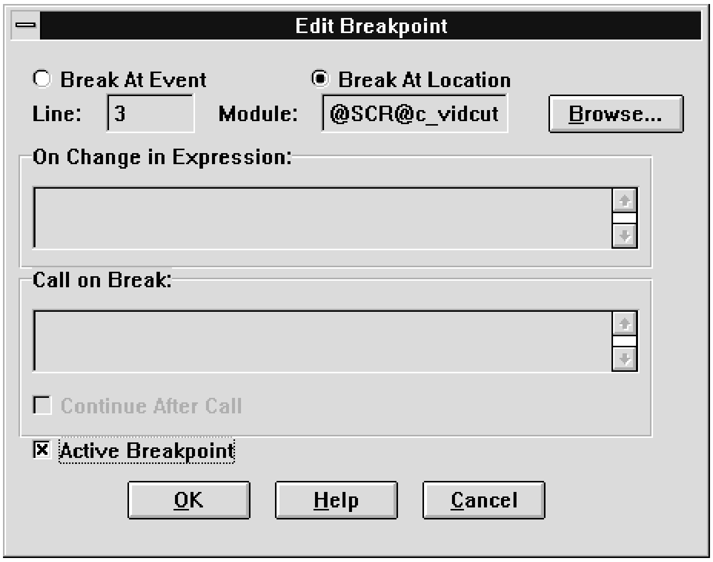
In general, it is easier to set location breakpoints by reading the code into the Source Code dialog box and double-click directly on the specific line of code to set the breakpoint.

Refer to page 39-16 for details on setting breakpoints in the Source Code window. Or, you can set a breakpoint from the Edit Breakpoint dialog box:

1. From the Edit Breakpoint dialog box, choose the Break at Location radio button.

2. Enter a line number and module at which to establish a breakpoint in the Line and Module fields, or choose Browse to select the module from the Open Source Module.

For instruction on how to identify source code from the Open Source Module, refer to page 39-12. Once you choose the module from Open Source Module, the Edit Breakpoint dialog box redisplays.


Edit Breakpoint

Break At Event [✓ Break At Location]

Line: 3 Module: @SCR@c_vidcut

On Change in Expression:

Call on Break:

☐ Continue After Call

☒ Active Breakpoint

Figure 39-16 Use the Edit Breakpoint dialog box in Location mode to set a code location breakpoint.

3. (Optional) Indicate by checking or unchecking the Active Breakpoint check box if the item should be added to the list of breakpoints in an active or inactive state.

4. Choose OK. The location breakpoint is added to the list of breakpoints (in the Breakpoints dialog).

Break on Change in Expression

You can direct the debugger, in expert mode, to make activation of a breakpoint dependent upon the value of an expression.

To create a breakpoint on an expression value (in expert mode):

1. Choose Breaks⟶Add or Edit. The Edit Breakpoint dialog box opens.

2. Enter a valid JPL expression in the On Change in Expression field. This causes the debugger to only stop at the event/location breakpoint (assuming it has been activated) if the value of the expression changes, that is, if the value changes from what it was when Trace⟶To Breakpoint was chosen.
3. Set break at Any Event if you want the debugger to awaken as soon as the change occurs.

To call a specified function (in expert mode):

From the Edit Breakpoints dialog box, enter the JPL or installed function to be executed in the Call on Break field. The debugger calls the specified function each time the breakpoint is reached.

To instruct the debugger to call the function without stopping execution, check Continue After Call.

Monitoring Variables and JPL Expressions

When running the debugger you can examine the contents of variables or expressions and observe runtime changes. Use the Data Watch window to inspect a property of the application, a screen, or widget, with a JPL expression. For more information on referencing Panther objects and properties, refer to page 19-33 .

To view data changes:

1. Choose View⟶Data Watch. The Data Watch window opens and displays the names of variables and expressions along with their current values (if defined).

   If variable is an array and you have defined as a range of values to watch, the variable is represented as

```
@range("
```

```
variable
```

```
",
 startOcc
,
endOcc). For example,
@range("title_id",2,4)
displays the title_id values associated with the second through
fourth occurrence of an array.
```

2. Enter the names of the variables or expressions you want to observe.

   The debugger displays the values of the variables/expressions in the Data Watch window, updating them as execution proceeds. The location of any variables specified is also identified.

To clear a variable/expression, delete it from the list. To clear all variables/expression, press `CLR`.

Modifying and Monitoring Application Data

Run the debugger in expert mode to take advantage of the data watching abilities available with the Application Data window.

To view and access data from any variable used in source code (in expert mode):

1. Access the desired source module (refer to page 39-12 for details on opening the Source Code window).
2. Position the cursor in the desired variable.
3. Choose Tools→Application Data to open the Application Data window.
4. From the Application Data window, you can:
   - Display the current value of specific variable. If the variable is an array, you can specify the range of occurrences to view.
   - Choose Break to set a breakpoint. The breakpoint is added to the list of breakpoints and occurs whenever the value of the variable or expression changes.
   - Choose Watch to add the variable or expression to the Data Watch window.

- Choose Modify to change the value of the variable.

CHAPTER

40 Identifying Users

An application must be able to identify its users and to prevent users without proper identification from its use.

Two-tier Applications

In two-tier applications, user authentication is needed for database access. When you connect to most database engines, you must supply a user name and password.

Panther applications use `DBMS DECLARE CONNECTION` statements to connect to a database engine. The documentation for each database engine lists the connection options available for that engine; refer to Database Drivers .

JetNet Applications

JetNet applications have two levels of identification:

- Connections to the middleware by the application users in the `client_init` command.

- Connections to the database by the application server using `DBMS DECLARE CONNECTION`.

MTS Applications

MTS offers two types of security for component packages: programmatic security, which uses interfaces to call within the application, and declarative security, which assigns users, or groups of users, to roles.

A role is the name assigned to a group of users that will access a component package. For example, a human resources application could define roles for Manager and Employee.

Panther MTS applications can implement programmatic security using the following functions which call methods of the `IObjectContext` interface:

- `sm_mts_IsCallerInRole`—Determines if a caller is assigned to a role, where caller is the identity of the process calling into the server.
- `sm_mts_IsSecurityEnabled`—Determines if security checking is enabled.

Refer to Microsoft's *MTS Documentation* for additional information about MTS package security.

CHAPTER

41 Optimizing Applications

Once the application is developed, you can optimize the performance in several key areas.

Database Fetches

The following items can affect the response time of database fetches:

- `dm_set_max_rows_per_fetch`—Sets the number of rows in each packet.
- Database indexes—Generate indexes, or additional indexes, on the database.
- Database access method—Instead of using the transaction manager, you can call stored procedures or SQL statements.
- Database locking method—In order to avoid having users waiting for database locks to be released, change to an optimistic locking scheme or deploy the application in a three-tier environment.

Web Applications

The following items can improve the performance of your Web application:

- Requester executable—Use the ISAPI or the NSAPI version of the requester executable, instead of the CGI version.
- Windows Web application servers—Use NTFS as the file locking system.

- Graphics—Place your graphics in the directory specified in the `ImageDir` variable to improve the loading of graphics files.

LDBs

The following items can affect the response time when using LDBs:

- Number of widgets in the LDB.

Part VII Deploying the Application

This section lists guidelines to follow when building application executables and preparing them for release.

CHAPTER

42 Building Application Executables

Several executables are provided to get Panther and Panther applications up and running. Depending on your application's architecture and the way Panther itself resides on your network, you will probably build and rebuild executables to meet a variety of development and deployment requirements.

There are executables (include `.exe` extension under Windows) used solely by the client:

- `prorun`—Invokes a production application.
- `prodev`—Invokes the development version of Panther.

And for JetNet/Oracle Tuxedo executables that are specific to the server:

- `proserv`—Invokes a Panther server.
- `prodserv`—Invokes a debuggable Panther server used for development.
- `progserv`—Invokes a conversion server specifically for use with applications that have been converted from two- to three-architecture.

For the most part, you can use the executables provided with your installation. However, if any of the following conditions apply, either at the outset or during development, or when you are ready to deploy your application, you need to build new executables:

- Under UNIX, to link in your database engine.
- To create a debuggable server executable for use with your database engine.
- Under Windows, if DLLs for your specific database engine are not included in your Panther installation and you or users of your application require direct (as opposed to remote) connections to the database.
- To incorporate user-written C modules.
- To include memory-resident components.
- Whenever you modify Panther's source code (for example, `jmain.c`).

This chapter describes:

- Steps for building Panther executables for UNIX and Windows.
- What to modify in Panther's source code so that you can create executables specifically for your application. For example, you can specify your application's start up screen or make files memory-resident.
- How to specify your own application icon for Windows.

Steps for Creating an Executable

The steps for building an executable assume that `SMBASE`, `SMVARS`, and `SMPATH` settings are the same as your development setup (for `prodev`) and are in effect.

Prepare the Application Directory

1. Create an application directory (if one doesn't already exist).
2. Copy all files from the distributed link directory to your application directory.
3. If you want to link your own C modules to the executable, copy them to your application directory.

Determine the Executables to Build

To specify which executables to build, either edit the makefile in your application directory or specify the target.

If you edit the makefile, make sure the appropriate client executables are commented or uncommented as needed (these are uncommented by default):

`PRORUN=prorun`	Runtime executable (`prorun32.exe` under Windows)
`PRODEV=prodev`	Development executable (`prodev32.exe` under Windows)

For JetNet and Oracle Tuxedo which support application servers, make sure the appropriate server executables are commented or uncommented as needed:

`PROSERV=proserv`	Server executable (uncommented by default)
`PRODSERV=prodserv`	Debuggable server executable (commented out by default)
`PROGSERV=progserv`	Conversion server executable (commented out by default)

Notes: To use Panther's debugger on an application server with your particular database, uncomment `PRODSERV`.

Link the Database Engine

You must link in your database engine to the executable if you are:

- Building UNIX executables, or
- The Windows installation did not include database DLLs for your engine.

Edit the makefile in order to identify the desired database for the executable. Uncomment the appropriate include statement for your particular database. The include statements are listed under the `SELECT DATABASE SOFTWARE` heading.

To link your database engine:

Uncomment the appropriate include statement for your database. You will also need to edit the file you uncomment (`makevars.dbs`) to indicate the correct version of your database software (refer to page 42-5 for details on editing `makevars.dbs`).

To exclude the JDB database from the executable:

Comment out the JDB include statement.

To link JDB and your database engine:

Leave the JDB database specification uncommented and uncomment the appropriate include statement for your database.

Include C Modules in the Executable

If your application uses user-written C modules, you'll want to link them into the application's executable.

To link your own C modules to the executable:

1. Copy the modules to the application directory.
2. Add the filenames to the `SRCMODS` macro of the makefile.

For example, if your application's C source code modules are `mymod1.c` and `mymod2.c`, change the `SRCMODS` line of the makefile to list the modules. For example:

UNIX: `SRCMODS = mymod1.o mymod2.o funclist.o`

Windows: `SRCMODS = mymod1.obj mymod3.obj funclist.obj`

When you include files within the executable, you don't have to include them in the deployed distribution.

Identify the Database Version

When you build an executable to include your database, once you edit the makefile, you must also ensure that the uncommented `makevars` file includes the appropriate information.

Refer to Database Drivers for your specific engine before changing these values.

To verify (or update) database-specific information:

Access the `makevars.dbs`, and ensure that the following settings are correct for your database engine:

- Case handling—Set the flag `dbs_INIT` to one of the following: d (default case conversion set), l (lowercase), u (uppercase), p (preserve case). The default is d. This specification deals primarily with the case conversion of database column names. Some database engines only create column names in a specific case or allow mixed cases. This case setting specification allows you to create all Panther variables in a particular case and Panther handles the conversion for you.

  Refer to the online *Database Drivers* for find out what the Panther default is for your database engine.

- Engine name—Set the flag `dbs_ENGNAME` to specify the default engine name. Refer to the online *Database Drivers* for your specific engine.

- Version—In the *`databaseName`* `PARAMETERS` section of the `makevars`, verify your database engine's version. Uncomment the appropriate block of parameters based upon this version. Also, verify and correct path names if necessary.

Compile the Changes

Once you have copy all required files to your application directory, and make the appropriate edits to the makefile (and optionally, to `makevars` and `jmain.c`), from the application directory, run the appropriate compiler utility at the command line: `make` under UNIX; `nmake` under Windows.

A compiling message and then the linking message are displayed. When the operating system returns you to the command prompt (with no errors), the executables have been built.

For more about makefiles, refer to your compiler's documentation on `make` or `nmake`.

If you are building executables for distribution, you can package them with the other required files for application release. Refer to page 43-1 for details.

Rename the Executables

The result of running the make or `nmake` utility is compiled executables ready for distribution. You can rename executables to reflect your application's name; there are no naming restrictions, except perhaps those imposed by your specific operating system.

Customizing Source Code for an Application

You can edit the source, `jmain.c`, for the Panther executable to change the default behavior of your application. jmain.c is located in the link directory and you should copy it to your application directory. Edit the source file prior to creating the executable if you want to do any of the following:

- Specify a startup screen for your application, such as, a welcome or login screen.
- Specify an icon for your application.
- Make Panther binary files memory-resident: screens, JPL modules, menus.
- Make configuration files used by Panther memory-resident.
- Rename the Panther library.

Whenever you edit `jmain.c` you must remember to rebuild your runtime executable (`prorun`) in order for changes to take effect.

Source Code Structure

`jmain.c` has three functions defined in it:

- `main`, `initialize` and `clean_up` which you can modify: `main` is defined globally and is the entry point to the entire application program. `main` calls the statically defined functions
- `initialize` and `clean_up`. Code necessary to your application can be inserted into the main routine. Any code inserted before `initialize` is executed before any Panther functions have been executed. `initialize` allocates internal data structures and sets the terminal characteristics. Code inserted after `initialize` is executed after Panther allocates internal data structures and sets the terminal characteristics, but before there are any screens and before there is a local data block.
- `clean_up` exits back to the operating system and restores the terminal's display state. Code called after `clean_up` is executed after all Panther functions have been executed.

If a finer granularity is needed, you can edit `initialize` and `clean_up` themselves. Do so only if you understand Panther thoroughly.

Notes: In general, you should not modify `jmain.c` to install event functions. You can declare most event functions in `funclist.c`. A few event function types, however, must be installed before or during initialization in `jmain.c`. For more information on event function installation, refer to page 44-4.

Specifying an Application Startup Screen

To identify a startup screen that will display when your application is first initialized, you must edit the `jmain.c` file you copied to your application directory.

Change the `start_screen_name` setting from 0 to the name of your top screen; for example, `start_screen_name = “login.scr”`; Include the specification after the comment for `TOPSCREEN`:

```
/* to hard code the name of the first screen, for */
/* example, “TOPSCREEN”, replace the following line */
/* with start_screen_name = "TOPSCREEN"; */
```

```
start_screen_name = "login.scr";
start_up (argc, argv);
```

Rebuild your runtime executable to have the change in `jmain.c` take effect.

Specifying an Application Icon

To ensure that an icon that will display when your application is installed on a Windows client machine, specify the name of the application icon in the `prorun.rc` resource file that you copied to your application directory.

In `jmain.c`, change the `ICONFILE` specification from `prolific.ico` to the name of your icon image.

Rebuild your runtime executable to have the change in `jmain.c` take effect.

Including Memory-Resident Components

All Panther components can be made memory resident: screens, menus, configuration files, and JPL modules. When a component is made memory resident, it means it is compiled as part of the executable. This can enhance application performance by minimizing its need to access the disk. It can also reduce the number of files in the distribution. Once these components are compiled into the executable, the individual components can then be eliminated from the distribution.

In function, memory residency means the component is loaded into memory when your application starts up. However, it's also important to recognize that your application would consume more memory at startup.

In general, the process of making a component memory-resident involves:

- Copying or extracting the desired component in its binary format from its library using `formlib`.
- Converting the component from binary format to a C data structure with the `bin2c` utility.
- Editing the source file, `jmain.c`, to include the desired C structure.
- Compiling the application to build the components into the application's executable.

Screens, Menus, and JPL

Panther creates binary files that are stored in Panther libraries. These include your screens and service components, JPL modules, menus, and any file created with and used by Panther (such as an application setup file named in `SMSETUP`). Memory-resident files are installed by way of a memory-resident list. The list is maintained by Panther and specifies the components you indicate in the source file.

To make screens, menus, or JPL modules memory-resident:

1. Extract the desired binary component from its library using `formlib`. Or run `jpl2bin` against the ASCII version of the component to get a binary file.
2. Convert the binary file to a C structure with `bin2c`, for example:

   ```
   bin2c filename.h filename.bin
   ```

3. In `jmain.c`, after the lines:

   ```
   #include "smdef.h"
   #include "smerror.h"
   #include "smuprdb.h"
   ```

   add the *`filename`*`.h` specification

   ```
   #include filename.h
   ```

4. In `jmain.c`, before the line:

   ```
   /* THE FOLLOWING FUNCTIONS ARE IN THIS MODULE */
   ```

   include the C structure:

   ```
   struct form_list mrforms[] =
   {
      "source_filename.ext", filename
        "",             (char *)0
   };
   ```

   All memory-resident files, modules, etc. should be listed, one per line, immediately before the line:

   ```
    "",             (char *)0
   ```

   This last entry in the list indicates the end of the list and is required.

5. In `jmain.c`, after the line:

   ```
   /* Place code...any Panther initializations here */
   ```

   add the following line:

```
sm_formlist(mrforms);
```

The call to `sm_formlist` dynamically adds the specified components in the `form_list` structure to Panther's internal memory-resident list.

6. Make sure the edited `jmain.c` source is in your application directory and create your application executable (refer to page 42-2, “Steps for Creating an Executable”).

Notes: Because the screen editor can only operate on library files, altering memory-resident screens and JPL during program development requires a cycle of test—edit—extract from library—reconvert with `bin2c`—recompile. Therefore you should make components memory-resident only at the very end of your development process.

At runtime, when Panther attempts to open a screen, it first looks in the memory-resident screen list for the requested screen; if found there, it's displayed from memory, while screens not in the list are sought in open libraries. In this way, you can open screens irrespective of their actual location—for example, with `sm_r_form`. You can later change the location of the screen without changing the calls to open them, simply by changing the memory-resident list.

filename extensions and case conversion

If you have specified the setup variable `SMFEXTENSION`, Panther appends its value to screen names that do not already contain an extension; take this into account when creating the screen list. For UNIX systems, Panther can also convert the name to uppercase before searching the screen list, as determined by the `FCASE` variable.

Alternatively, if you are using a custom executive, `sm_d_form` and related library functions can be used to display memory-resident screens; each takes as one of its parameters the address of the global array containing the screen data, which usually have the same name as the file in which the original screen was originally stored.

Configuration Files

Any or all of the required configuration files as well as optional configuration files used by Panther can be made memory-resident. These files include:

- Message files (`msgfile.bin`)
- Video file (`*vid.bin`)
- Configuration map file (`*cmap.bin`)
- Key translation file (`*keys.bin`)

To make configuration files memory-resident:

1. Copy the desired binary files from the `config` directory or extract (using `formlib`) them from the common library (if they are already associated with your application).
2. Convert the binary file to C structure with `bin2c`, for example:

   ```
   bin2c filename.h filename.bin
   ```

3. In `jmain.c`, after the lines:

   ```
   #include "smdef.h"
   #include "smerror.h"
   #include "smuprdb.h"
   ```

   add the `filename.h` specification:

   ```
   #include "filename.h"
   ```

4. In `jmain.c`, after the line:

   ```
   /* Place code to...any Panther initializations here */
   ```

   add the following line or lines, specific to each configuration file type:

   ```
   sm_n_msg_read ("SM", SM_MSGS, MSG_MEMORY|MSG_INIT,
   msgfile);
   sm_vinit (videofile);
   sm_load_colormap (cmapfile);
   sm_keyinit (keyfile);
   ```

If a configuration file is memory-resident, the corresponding environment variable or SMVARS entry is unnecessary and the binary files needn't be included in the common library of your distribution.

Rename the Distributed Panther Library

The distributed library, `prorun5.lib`, contains the support routines and screens used by Panther. It is required in your application's distribution as well. In addition to renaming executables, you can also rename this library to conform to your application's name.

To rename the Panther library:

1. Copy `prorun5.lib` from the `config` directory to your application directory and rename the copied file.
2. In `jmain.c`, replace the line:

   ```
   if ((lib = sm_l_open (p = RUNTIME_LIBRARY)) < 0)
   ```

   with:

   ```
   if ((lib = sm_l_open (p = "myapp.lib")) < 0)
   ```

Subsystem Installation

After the definition of the main function, there is a `JTERM_COMPRESSION` subsystem macro definition. This subsystem increases the communication efficiency and execution speed for applications when they are accessed by the Panther terminal emulator. It increases the application's memory requirements. By default, it is set to 0. To turn the subsystem on, set the macro to 1.

Oracle Tuxedo Executables

You need to update the Oracle Tuxedo XA RM file before you can use the provided makefile to relink executables. This is done by appending the contents of `$SMBASE/samples/Oracle Tuxedo/RM` to `$TUXDIR/udataobj/RM` and confirming the appropriate section for your database and version.

CHAPTER

43 Preparing Applications for Release

Before delivering an application to its users, you need to package it for delivery. There are some basic constructions to remember that will make packaging your application more modular. This chapter describes general information about:

- The steps to be completed for deployment.
- Which files are needed for distribution.
- What customizations you can make.

The list of files that you must include in a distribution varies depending on the application's components and the platform or platforms on which it runs. In addition, there are variations in distribution depending on your application's architecture. The documentation includes checklists for the major architectures:

- Two-tier applications—See Appendix D, “Deployment Checklist for Two-tier Applications,” in this manual.
- JetNet and Oracle Tuxedo applications—See Appendix E, “Application Setup Checklist,” in the *JetNet/Oracle Tuxedo Guide*.

Basic Deployment Steps

How to Deploy your Application

1. Create application executables (refer to page 42-1).

2. Make sure all binary files—screens, JPL modules, menu bars/toolbars—used by your application are stored in libraries (refer to page A-14 for details on using the `formlib`).

   Your Panther application consists of at least one to three Panther libraries, depending on the application's architecture: a library for storing the objects that define the user interface (`client.lib`); a library for storing the objects required by application servers (`server.lib`); and, for JetNet/Oracle Tuxedo, a common library (`common.lib`) for the objects that are needed by both client and server.

   If, however, the components are to be made memory-resident, extract the desired components from their respective libraries before building your executables.

3. Make sure all graphical elements referenced, via property settings, by your Panther screens, such as pixmaps and icons, are included in a client library.

   For web applications, alternatively, you can place your graphics files in the `htdocs` directory on your HTTP server and set the `ImageDir` variable in the web initialization file to point to that location.

4. When the libraries are complete, use `formlib -o` to lock the libraries. This step is irreversible. (After locking the library, you must use `formlib -x` to extract the contents and `formlib -c` to create a new library for editing or `formlib -m` to condense and unlock the library).

5. Create an application directory and copy all of the required files, libraries, and subdirectories to this directory.

6. Make sure the environment variables `SMBASE`, `SMVARS`, `SMTERM` are set correctly:

- UNIX:

    - `SMBASE` should point to your application directory (for example, `setenv SMBASE /usr/myapp`).
    - `SMVARS` should point to `smvars.bin` in your `config` subdirectory of your application directory (for example, `setenv SMVARS /usr/myapp/config/smvars.bin`).
    - `SMTERM` should define the platform (for example, `setenv SMTERM X`).
- Windows:
    - Set `SMVARS` to point to `smvars.bin` in the `config` subdirectory of your application directory. The variables can be defined in your application's initialization file (`prol5w32.ini`) or in `autoexec.bat`.
    - (Optional) Set `SMPATH` to point to other application directories.
    - (Optional) Set the directory for your application's startup icon to the application directory.

Note: Remember to run `var2bin` if you edit the `smvars` file. Then include the newly compiled `smvars.bin` in your distribution.

7. Make sure the appropriate libraries are open on startup of your application, set the `SMFLIBS` variable (in `smvars` or in the initialization file for clients).

   On Windows, libraries are delineated with a `;` or `|`.

   On UNIX, libraries are delineated with a `:` or a `|`.

8. JetNet/Oracle Tuxedo:
    - Create or copy the JetNet configuration file (refer to the *JetNet Guide/Oracle Tuxedo Guide* for details). For Oracle Tuxedo applications, create or copy your Oracle Tuxedo configuration file.
    - Make sure connections to the middleware are defined in the setup variables `SMRBCONFIG` on local UNIX clients and `SMRBHOST` and `SMRBPORT` on Windows and remote clients.
    - Make sure the appropriate libraries are open on startup for the servers in `proserv.env`, `progserv.env`, and `machine.env` using `SMFLIBS`.
9. COM/DCOM/MTS:
    - For COM: on each application client, copy the DLLs (and type libraries) to the application directory and register the COM component.

- For DCOM: on the machine that is the COM component server, copy the DLLs (and type libraries) to the application directory and register the COM component. On the application clients, run the client registration file.
- For MTS: on the server running MTS, copy the DLLs (and type libraries) to the application directory and install the COM components in the Microsoft Management Console. Create an export file for each component package and have the application clients run that file.

10. Then run your application's executable.
11. Test the application.
12. And ship it!

Required Files

All Panther applications require the following files:

- Panther executables (refer to page 42-2 for details on creating your application's executables).
- Library or libraries containing application components that define the user interface. `client.lib` is the suggested name, but another common practice is to create libraries based on their content, such as `images.lib` for graphics files.
- Panther library (`prorun5.lib`) containing all screens and JPL used by Panther.
- The following files placed in one of the application libraries:
  - Panther message file (`msgfile.bin`).
  - Keyboard files for the platforms on which your application will run; for example, if your Panther application runs on `xterm`, copy `xterm` keys.bin to your distribution directory.
  - Configuration map file for the desired platform; for example, copy `wincmap.bin` for Windows or `xwincmap.bin` for Motif.

  - Character-mode platforms need the video files for the platforms on which your application will run.

- Motif resource files, for example, the `Prolifics` resource file.
- Windows-specific initialization files, for example, `prol5w32.ini`.
- Windows-specific Dynamic Link Libraries (DLLs). All `*.dll` files in the distributed `util` directory except for `dtext.dll`, `cgmzv.dll`, and `ctl3d.dll` which are only used by the online help system.
- Setup information in the form of `smvars.bin`, or specified in the system's initialization file.
- JetNet and Oracle Tuxedo applications also require the following files:
  - JetNet shared libraries for UNIX (for example, `lib/*.so.*` on Sun).
  - JetNet administrative executables. All files in the distributed `bin` directory.
  - The contents of the distributed `locale` and `udataobj` directories.
  - Panther utilities (for example, `jetman`, `rbboot`, `rbshutdown`).
  - JetMan resource file for Motif (`Jetman`) or initialization file for Windows (`jetman.ini`).
  - Library (`common.lib`) containing application components used by the client and server for JetNet and Oracle Tuxedo applications.
  - Library (`server.lib`) or libraries containing service components and service-related objects that define the server.
  - Environment definition files* for application servers (*`.env` files)
- COM/MTS applications also require the following files in the application directory:
  - Panther DLLs for the COM component server.
  - For each service component, a DLL (`.dll`).

Optional Files

The following files are optional, depending on your application's components, requirements, and how it is configured:

- Graph requirements. If your application displays or uses business graphs and charts, include `grafcap`, `symbold`, and `symbols1` in your distribution. Include any graph-specific font files that are used in your application (for example, `aldine.fnt`, `centry.fnt`, `duplex.fnt` from the distributed `config` directory).

  You also need the following platform-specific files:

  - Under UNIX: the executables, `gdsp` and `swsdrvr`.
  - Under Windows: `libsti.ini` (belongs in the Windows directory), `libsti32/64.dll` (copy to your application directory).

- LDB screens. If you have any LDB screens that are used for data transfer or initialization (usually named `ldb.scr`), include them in the appropriate library.
- Message file. If you have stored messages in message files, they need to be in binary format and copied to the appropriate library.
- Graphics files. Images that are defined as pixmap properties to display on buttons and screens in your application. Include these in the client library, and for web applications, in the `htdocs` directory.
- ActiveX controls (for Windows and Web applications). ActiveX controls in Web applications should be digitally signed and used in accordance with the control's licensing scheme. Since Windows applications will not be deployed on a Web browser, it is not necessary to digitally sign the control; however, your application's installation program should install your ActiveX control and register the control on the user's system.

Specifying Files and Directories

For the client portion of your application, you can use the `smvars` file to specify all the libraries and directories that Panther needs to run. The only variables that must be set by the user would be `SMTERM` and `SMVARS`, which tell Panther which binary configuration files to use and where to find the application's files. Refer to page 2-1 in the *Configuration Guide* for more information about setting up the system environment.

Alternatively, your application can specify required files through calls to Panther functions:

- `sm_keyinit` initializes the key translation file from the specified key file.
- `sm_vinit` initializes the video translation file from the specified video file.
- `sm_soption` supplies the search path for Panther binaries.
- `sm_msg_read` reads a message file into memory.
- `sm_l_open` opens a Panther library.

Detailed information on each function and their variants is available in the *Programming Guide*.

Customizing the Distribution

This section describes additional measures you can take when you package your application. To ensure that these changes take effect, it is best to build your executable after customizing the appropriate files.

Configuration Support

For JetNet/Oracle Tuxedo applications, if you are administering the setup of your deployed application, you can provide a middleware configuration file that is similar to the one built when the application was being developed. If the application is being setup by your customer, you can provide documentation for using the JetNet manager and/or scripts for defining the number of client workstations, the number of machines, and the number of application servers that will be used to run your application.

Specifying a Startup File

When your application starts up it uses an initialization file (`prol5w32.ini` or `prol5w64.ini`) under Windows and a resource file (`Prolifics`) under Motif. For Motif, instead of using the distributed files, you can change the name to reflect your application's name and specify the file in `piinit.c` (found in the `link` directory). To include the name of your resource file, for example:

```
sm_pi_xm__setup("myAppSetup");
```

For Windows, you can specify the name of the ini file on the Start Menu command line. For example, the `PlayMusic` application would place `PlayMusic.ini` in the Windows directory and call that ini file with the following command setting:

```
C:\Prolifics\Panther\prorun.exe -ini PlayMusic
```

Specifying a Title Screen

You can design your own splash screen, just like the one that displays when you startup the screen editor, that will display when your application is initialized. The screen can be a BMP, JPEG, or GIF format. You specify the name of the screen in your startup file:

- For Motif: In your resource file (`Prolifics`), add the line:

  ```
  Prolifics*introPixmap:  "titleScreen.ext"
  ```

- For Windows: In your initialization file (`prol5w32.ini`), include the name on the line:

  ```
  IntroPixmap="titleScreen.ext"
  ```

Specifying Your Own Icon

When you minimize your application in Windows, the default Panther icon is used. You can replace the default icon with one of your own. Edit the `prorun.rc` file located in the `link` directory to include the name of your `.ico` file, for example:

```
#define ICONFILE myIcon.ico
```

Part VIII Advanced Development Topics

This section lists some advanced development topics:

CHAPTER

44 Installed Event Functions

Installed event functions are functions that are written in C and are called at specific events during program execution. Panther recognizes many different stages of program execution as events that can invoke functions—for example, screen entry and exit, widget entry and exit, and client initialization.

All installed functions must be compiled into the application so that Panther can find and execute them at the proper time. Most Panther library functions are already installed for immediate access during the design process.

This chapter shows how to write event functions in C and install them in your application. For information about Panther events and event processing, refer to Chapter 17, “Understanding Application Events.”

Installed Function Types

Installed functions can be divided into two general types: those that are called explicitly and those that are called automatically on specific stages of program execution:

- Demand functions are explicitly called from a Panther component through one of its event properties, such as a field's Exit Function property. Demand functions can also be called by JPL using the `call` command.
- Automatic functions execute on all occurrences of an event type. Each automatic function is identified with a single event type. For example, you can install an automatic screen function that executes on entering and exiting all screens. These functions are never explicitly called in the application code or screens; instead, they are called automatically at the appropriate stage of program execution.

Panther can call automatic and demand functions for the same object. For example, on screen entry, the automatic screen function is always called if one is installed. If a screen's Entry Function property also specifies a function, this function is called and executed, too.

Demand Functions

A demand function can be called by name from event properties of any component in a Panther application: groups, screens, menu items, logical keys, and JPL modules. Except for library JPL modules, function calls are stored with the application's screens and can be edited through the screen editor. For example, each widget's properties window lets you specify entry, exit, and validation functions.

Demand functions usually perform tasks that are specific to their callers. For example, you might write an exit function for a widget whose data requires special validation. You then specify this function through the widget's Validation Func property. At runtime, the function is invoked when the user tabs out of the widget.

You can also write demand functions in a JPL module and make the module available to the application through the public command. You can then call that module's procedures—for example, as a widget's entry function, or from a control string; Panther executes the JPL code if no C function of the same name is installed. For more information about JPL, refer to Chapter 19, “Programming in JPL.”

Automatic Functions

Automatic functions are called automatically at specific event types. For example, an automatic screen function executes anytime a screen opens and closes. Unlike demand functions, automatic functions are independent of any one widget, screen, or other application component. Automatic functions cannot be written in JPL. However, you can call a JPL procedure from an automatic function through `sm_jplcall`.

In general, an application can have only one automatic function of each type installed at a time. Thus, there can be only one automatic screen function, one insert toggle function, and so on. Timeout and timer functions are the exception among automatic functions: you can install multiple timeout and timer functions, where each one is called when its own interval expires.

Standard versus Non-standard Arguments

Panther automatically supplies a fixed number of arguments for all installed function types except those that are installed as type `PROTO_FUNC` (prototyped functions). Arguments that are automatically supplied by Panther are called standard arguments. For example, screen functions get two standard arguments: the screen's name, and a bitmask that tells when and how this function was called. If you use these arguments, you must ensure that function definition parameters correspond in number and type to those supplied by Panther.

You can also write functions whose arguments are explicitly supplied by the application. These functions must be installed as prototyped functions. Panther expects calls to any functions thus installed to supply their own arguments.

Installation

Most functions are typically installed in the source file `funclist.c`. The coding required to install a function consists of two steps:

1. Prepare the function for installation by including it in a `fnc_data` structure. If the function type allows installation of multiple functions, declare an array of `fnc_data` structures, where each data structure specifies a function.

2. Install the function with a call to `sm_install` in the `sm_do_uinstalls` function.

The following sections describe each of these steps.

Notes: For greater efficiency, prototyped function declarations should be `#include`'d in `funclist.c`.

Preparing Functions for Installation

Before you can install a function, you must first include it in a `fnc_data` structure. The following statements prepare two prototyped functions and one automatic screen function for installation:

```
struct fnc_data proto_list[] = {
   SM_INTFNC ( "mark_flds(i,i)", mark_flds ),
   SM_INTFNC ( "report(s,s)", report )
};

struct fnc_data autosc_struct = SM_OLDFNC( 0, auto_sfunc);
```

Each `fnc_data` structure is initialized with the following information:

SM_*FNC Macro

Prefix a `fnc_data` structure with one of several macros that determines the function's return type and whether it dereferences its arguments. Use one of these macros:

- `SM_INTFNC` specifies that the function dereferences arguments supplied from JPL and returns an integer value.
- `SM_STRFNC` specifies that the function dereferences arguments supplied from JPL and returns a string value.
- `SM_DBLFNC` specifies that the function dereferences arguments supplied from JPL and returns a double precision value.
- `SM_ZROFNC` specifies that the function dereferences arguments supplied from JPL and always returns 0 to its caller.
- `SM_OLDFNC` specifies that the function does not dereference JPL-supplied arguments and returns an integer value. Use this macro for all non-prototyped functions and for any function written for pre-Panther applications.

Function Name

The first value of a `SM_*FNC` macro specifies the function's name. Names of prototyped functions must include their argument types. In the previous example, `mark_flds` takes two integer arguments, while `report` takes two strings.

If the function type allows installation of only one function, supply 0.

Strings and integers are the only two data types that can be passed. If the value is not an integer, pass the value as a string and do the appropriate conversions in your code.

Function Address

The second value of a `SM_*FNC` macro is the address of the function—that is, its C identifier.

Installing Functions

You install functions through the library function `sm_install`. For example, given the earlier `fnc_data` structures, these statements install the functions in `proto_list` and `autosc_struct`:

```
int ct = sizeof ( proto_list ) / sizeof ( struct fnc_data );

sm_install ( PROTO_FUNC, proto_list, &ct ) ;
sm_install ( DFLT_SCREEN_FUNC, &autosc_struct, (int *)0 ) ;
```

This function takes three arguments:

func_type

Specifies the function type, one of the constants in Table 44-1. In this table, function types are divided into two groups: those that allow installation of multiple functions; and those that allow installation of only one function. Each type is discussed later in this chapter.

Table 44-1 Installed function types

Multiple function installation	
SCREEN_FUNC	Screen Functions
FIELD_FUNC	Field Functions
GRID_FUNC	Grid Functions
GROUP_FUNC	Group Functions
CARD_FUNC	Tab Control Functions
PROTO_FUNC	Prototyped Functions
TIMER_FUNC	Timer Functions
TIMEOUT_FUNC	Timeout Functions
CONTROL_FUNC	Control Functions
TP_INITDATA_FUNC	Client Authentication Functions
TP_INITPOST_FUNC	Client Post-Connection Functions
Single function installation	
DFLT_SCREEN_FUNC	Default Screen Function
DFLT_FIELD_FUNC	Default Field Function
DFLT_GROUP_FUNC	Default Group Function
KEYCHG_FUNC	Key Change Function

Table 44-1 Installed function types *(Continued)*

ERROR_FUNC	Error Function
INSCRSR_FUNC	Insert Toggle Function
EXTERNAL_HELP_FUNC	Help Function
CKDIGIT_FUNC	Check Digit Function
UINIT_FUNC	Initialization Function
URESET_FUNC	Reset Function
RECORD_FUNC	Record Function
PLAY_FUNC	Playback Function
STAT_FUNC	Status Line Function
VPROC_FUNC	Video Processing Function

funcs

The name of the `fnc_data` structure that includes the functions to install.

num_funcs

The address of a variable that contains the number of functions included in funcs. If funcs is an array of `fnc_data` structures, get the number of functions declared in the array before calling `sm_install`. If the function type allows installation of only one function, supply a null integer pointer—(int *)0.

For example, this statement gets the number of functions installed in the `fnc_data` array `proto_list`.

```
int pct = sizeof ( proto_list ) / sizeof ( structfnc_data );
```

Prototyped Functions

Prototyped functions are functions that get only the number and type of arguments that you specify. Prototyped functions are demand functions—that is, they must be specified by name from a Panther component, such as a widget or screen. Prototyped functions can also be called or referenced in JPL via commands such as `call` or `service_call`.

All prototyped functions are installed together in their own function list. When a prototyped function is called, it is supplied the arguments that you specify instead of the standard arguments otherwise supplied by its caller. Thus, if a screen entry event calls a function, and Panther finds this function on the list of prototyped functions, Panther passes the arguments that follow the function's name instead of the two standard arguments otherwise supplied to a screen function. As developer, you must make sure that prototyped function calls supply the correct number and type of arguments.

You can specify prototyped function calls through the screen editor. For example, the screen properties window lets you specify prototyped functions for screen entry and exit. Prototyped functions can also be called in JPL procedures.

Accessing Standard Argument Information

Although prototyped functions that are called by widgets and groups do not get standard arguments, Panther has several library functions that let you get equivalent information about a widget or group.

`sm_inquire` can return a widget's number, validation state, and occurrence number, and a group's validation state, according to the argument that you supply:

Argument	Return Value
`SC_AFLDNO`	Number of the widget calling a prototyped function. Corresponds to the first standard argument supplied to a widget function.

Argument	Return Value
SC_AFLDMDT	Bit mask that indicates the widget's validation state and why the function was called. Corresponds to fourth standard argument of a widget function.
SC_AFLDOCC	Occurrence number of the widget that called the function. Corresponds to the third standard argument of a widget function.
SC_AGRPMDT	Bit mask that indicates the group's validation state and why the function was called. Corresponds to the second standard argument of a group function.

You can get the second standard argument of a widget function, a pointer to a copy of the widget's contents, through `sm_*getfield`§ or `sm_*fptr`.

You can also get the first standard argument of a group function, a pointer to the group name, through `sm_getcurno` and `sm_*ftog` at group entry and exit. Access to the group name at group validation is not supported because the group might be undergoing validation as part of screen validation.

Prototyped functions cannot access the standard arguments of a screen. If a function requires this information, you should install it as a demand or automatic screen function.

Installing Prototyped Functions

Prototyped functions are listed with their argument types as members of a `fnc_data` data structure. The list of argument types is enclosed in parentheses: Panther supports string and integer arguments, specified by `s` and `i`, respectively. The following declarations and definitions support the installation of two functions:

```
struct fnc_data pfuncs[] = {
   SM_INTFNC ( "mark_flds(i,i)", mark_flds ),
   SM_INTFNC ( "report(s,s)", report )
};

int pfuncs = sizeof ( pfuncs ) /
             sizeof ( struct fnc_data ) ;
```

In this example, `marks_flds` is prototyped to take two integer arguments and `report` takes two string arguments.

The macro `SM_INTFNC` specifies that the function dereferences its arguments and returns an integer value. For string returns, substitute `SM_STRFNC`; for double precision returns, substitute `SM_DBLFNC`.

Panther supports any combination of strings and integers from zero to five arguments, and functions with six integer arguments. If a function's arguments do not conform to these requirements—for example, there are more than six, or they include an unsupported data type—you can call it indirectly through a wrapper function.

The following library call to `sm_install` installs these functions.

`sm_install` is usually called in `sm_do_uinstalls`, found in the source module `funclist.c`:

```
sm_install( PROTO_FUNC, pfuncs, &pcount );
```

Screen Functions

You can install an automatic screen function that is called on screen entry and exit. You can also install one or more demand screen functions that can be called explicitly at different stages of program execution. Both automatic and demand screen functions get arguments that describe the screen's current state.

Panther executes the automatic screen function on both entry and exit for that screen. On entry, the automatic screen function is executed before the screen's entry function. If a screen has a JPL module, its unnamed procedure is executed on screen entry before the automatic function. On exit, the screen's exit function is executed before the automatic screen function.

Panther optionally recognizes overlay and reexposure of a screen as exit and entry events, respectively. This depends on how Panther setup variable `EXPHIDE_OPTION` is set. If the variable is set to `ON_EXPHIDE`, screen exit and entry `functions` are invoked on screen overlay and reexposure. Overlay of a screen can occur because another screen opens or is selected; reexposure can occur because an overlying screen closes or is deselected.

Notes: It is not advisable to open a screen from a screen entry function, since such an event yields undefined results.

Screen Function Arguments

All screen functions receive two arguments in this order:

- A pointer to a null-terminated character string that contains the screen's name.
- An integer bitmask that indicates the screen's current state and why the function was called.

The second parameter can have one or more of the following flags set:

K_ENTRY

The function was called on screen entry.

Equivalent:

```
if(param2 & K_ENTRY)
```

K_EXIT

The function was called on screen exit.

Equivalent:

```
if (param2 & K_EXIT)
```

K_EXPOSE

The function was called for one of these reasons:

- The screen was selected.
- The screen was deselected.

- The screen is hidden because a window popped over it—`K_EXIT` and `K_EXPOSE` are set.
- The screen is reexposed because a window that overlay it closed—`K_ENTRY` and `K_EXPOSE` are set.

Equivalent:

```
if (param2 & K_EXPOSE)
```

K_KEYS

Mask for the bits that indicate which event caused the screen to exit. You should test the intersection of this mask and the second parameter against `K_NORMAL` or `K_OTHER`.

K_NORMAL

A "normal" call to `sm_close_window` caused the screen to close.

Equivalent:

```
if ((param2 & K_KEYS) == K_NORMAL)
```

K_OTHER

The screen closed because another form is displayed or because `sm_resetcrt` is called.

Equivalent:

```
if ((param2 & K_KEYS) == K_OTHER)
```

Screen Function Returns

Screen functions should return 0 if they do not reposition the cursor or change the screen. If a screen function does move the cursor, it should have a non-zero return value, which prevents `sm_input` from repositioning the cursor.

Installation of an Automatic Screen Function

You can install only one function as the automatic screen function. The following statement, typically found in `funclist.c`, includes the automatic screen function `auto_sfunc` in the `fnc_data` structure `autoscr_struct`. To see the code for this function, refer to page 44-62.

```
struct fnc_data autoscr_struct = SM_OLDFNC( 0, auto_sfunc) ;
```

The following line of code, typically found in the function `sm_do_uinstalls` in funclist.c, installs `auto_sfunc` as the default screen function:

```
sm_install ( DFLT_SCREEN_FUNC, &autoscr_struct, (int *)0 ) ;
```

Installation of Demand Screen Functions

You can install multiple functions as demand screen functions. The following statements, typically found in funclist.c, include two all-purpose screen entry and exit functions `sEntry` and `sExit` in the `fnc_data` structure `sfuncs`:

```
struct fnc_data sfuncs[] =
{
   SM_OLDFNC( "sEntry", sEntry ),
   SM_OLDFNC( "sExit", sExit ),
};
```

```
int scount = sizeof ( sfuncs ) / sizeof ( struct fnc_data ) ;
```

The following line of code, typically found in the function `sm_do_uinstalls` in `funclist.c`, installs the functions in `sfuncs` as demand screen functions:

```
sm_install ( SCREEN_FUNC, sfuncs, &scount ) ;
```

Field Functions

You can install an automatic field function that is called on field entry, exit, and validation. You can also install one or more demand field functions that can be called explicitly at different stages of program execution. Both automatic and demand field functions get arguments that describe the field's current state.

Panther executes the automatic field function on field entry, exit, and validation. You can install an automatic field function that is invoked on any of these events for all fields. You can separately install demand field functions, which individual fields can explicitly invoke for entry, exit, or validation. These functions are installed in the field function list; a field specifies one of these through its properties window as its entry, exit, or validation function.

Automatic field functions can access non-standard information for specific fields through the field's memo text properties. For an example, see page 44-65.

Execution

Panther executes the automatic field function on all field events. On entry, the automatic field function is executed before the field's entry function. On exit, Panther first calls the field's validation function, then its exit function, and finally the automatic field function. If the field has JPL validation, this module is executed after the validation function.

Entry

Panther can recognize two events as field entry: when the cursor enters a field; and when the screen's current field is reactivated because an overlying window closes, if setup variable `EXPHIDE_OPTION` is set to `ON_EXPHIDE`.

Notes: It is not advisable to bring up a dialog box, such as a message dialog, from a field entry function, since opening a screen between a mouse down and a mouse up event yields undefined results.

Exit

Panther can recognize two events as field exit: when the cursor leaves a field; and when a window overlays the field's screen, if setup variable `EXPHIDE_OPTION` is set to `ON_EXPHIDE` (the default).

Validation

Validation functions are called under the following conditions:

- As part of field validation, when you exit the field or scroll to the next occurrence by filling it, or by pressing `TAB` or `NL`. Widget functions are called for validation only after the field's contents pass all other validations for the field.

  `BACKTAB`, arrow keys, and mouse clicks outside the field trigger field validation only if the setup variable `IN_VALID` is set to `OK_VALID`. The default setting is `OK_NOVALID`
- When the application code calls `sm_fval` or `sm_s_val` to force field or screen validation.
- As part of screen validation when `XMIT` is pressed. By default, this logical key is mapped to the function `sm_s_val`, which validates all fields on the screen. The setup variable `XMIT_LAST` can also be set so screen validation occurs when `TAB` and/or NL are pressed in a screen's last field.

If a field calls a validation function and belongs to a menu, radio button group, or check box group, this function is called when the field is selected. The validation function of a check box is also called when the field is deselected.

Field Function Arguments

All field functions receive four arguments in this order:

- An integer that contains the field's number.

- A pointer to a null-terminated character string that contains a copy of the field's contents.
- An integer that contains the occurrence number of the data.
- An integer bitmask that indicates the field's validation state and why the function was called.

The last parameter can have one or more flags set. The following sections describe these flags:

VALIDED

The field has passed validation and remains unmodified. Note that Panther always calls field functions for validation whether or not the field already passed validation. You can test this flag and the MDT flag to avoid redundant validation. This flag setting can also be accessed and modified through the field's valided property.

Equivalent: `if(param4 & VALIDED)`

MDT

The field's data changed since the current screen opened. If the screen entry function modifies the field's data when the screen opens, the MDT flag remains unset. However, if the same screen entry function executes because the screen is reexposed—for example, through closure of an overlying window—modification of the field's data sets the MDT flag. This flag setting can also be accessed and modified through the field's mdt property.

Panther never clears this flag once it is set, unless you explicitly clear it by setting the field's mdt property to `PV_NO`, or clear all fields on the screen by calling `sm_cl_all_mdts`.

Equivalent: `if(param4 & MDT)`

K_ENTRY

The field function was called on field entry.

Equivalent: `if(param4 & K_ENTRY)`

K_EXIT

The field function was called on field exit. Note that if neither `K_ENTRY` nor `K_EXIT` are set, the field is undergoing validation.

Equivalent: `if(param4 & K_EXIT)`

K_EXPOSE

The field function was called because a window overlying this field's screen opened or closed:

`K_EXPOSE` and `K_ENTRY are set: the overlying window closed and the field is exposed.` `K_EXPOSE` and `K_EXIT` are set: the overlying window opened and the field is hidden.

Equivalent: `if(param4 & K_EXPOSE)`

K_EXTEND

The field is an extended selection list box.

K_EXTEND_LAST

For extended selection list boxes, the field is the last item in the list box.

K_KEYS

Mask for the flags that tell which keystroke or event caused field entry, exit, or validation. The intersection of this mask and the fourth parameter to the field function should be tested for equality against one of the next six flags.

K_NORMAL

A "normal" key caused the cursor to enter or exit the field in question. For field entry, "normal" keys are `NL`, `TAB`, `HOME`, and `EMOH`. For field exit, only `TAB` and `NL` are considered "normal."

Equivalent: `if((param4 & K_KEYS)==K_NORMAL)`

K_BACKTAB

The `BACKTAB` key caused the cursor to enter or exit the field.

Equivalent: `if((param4 & K_KEYS)==K_BACKTAB)`

K_ARROW

An arrow key caused the cursor to enter or exit the field.

Equivalent: `if((param4 & K_KEYS)==K_ARROW)`

K_SVAL

The field is being validated as part of screen validation.

Equivalent: `if((param4 & K_KEYS)==K_SVAL)`

K_USER

The field is being validated directly from the application with `sm_fval` or `sm_validate`.

Equivalent: `if((param4 & K_KEYS)==K_USER))`

K_OTHER

A key other than `BACKTAB`, an arrow key, or a "normal" key caused the cursor to enter or exit the field. For field entry, "normal" keys are `NL`, `TAB`, `HOME`, and `EMOH`. For field exit, only `TAB` and `NL` are considered "normal."

Equivalent: `if((param4 & K_KEYS)==K_OTHER)`

K_INSDEL

The `INSL` or `DELL` key caused field exit or entry. Use this flag in field exit or entry processing to determine whether the entry or exit event resulted from the user inserting or deleting an occurrence. For example, the following code differentiates between field exit events that are caused by `INSL` or `DELL` inserting or deleting an occurrence, and other actions that might cause field exit, such as the user pressing `TAB`:

```
if (param4 & K_EXIT)
{
   if((param4 & K_KEYS) == K_INSDEL)
   // if exit occurred because user inserted or deleted
   // occurrence, do nothing
   {
      return
   }
   // 'real' exit occurred, so perform exit processing
   ...
}
```

Field Function Returns

Field functions called on entry or exit should return 0 if they leave the cursor position unchanged. Field functions called for validation should return 0 if the field contents pass the validation criteria. A non-zero return code in a validation function should indicate that the field does not pass validation.

If the returned value from a field function is 1, the cursor position remains unchanged. Any other non-zero return value repositions the cursor to the field. This repositioning is useful when an entire screen is undergoing validation, because the field that fails validation might not have the cursor in it. It is generally good design practice to use the field validation function to reposition the cursor before you display an error message. This reinforces the link between the error message and the offending field.

Installation of an Automatic Field Function

You can install only one function as the automatic field function. The following statement, usually found in `funclist.c`, includes the automatic field function `auto_ffunc` in the `fnc_data` structure `autofld_struct`. To see the code for this function, refer to page 44-65.

```
struct fnc_data autofld_struct = SM_OLDFNC( 0, auto_ffunc ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in funclist.c, installs `auto_ffunc` as the default field function:

```
sm_install ( DFLT_FIELD_FUNC, &autofld_struct, (int *) 0 ) ;
```

Installation of Demand Widget Functions

You can install multiple functions as demand field functions. The following statements, usually found in `funclist.c`, include two all-purpose field entry and validation functions `fentry` and `fvalid` in the `fnc_data` structure `ffuncs`. To see the code for these functions, refer to page 44-69.

```
struct fnc_data ffuncs[] =
{
   SM_OLDFNC( "fentry", fentry ),
   SM_OLDFNC( "fvalid", fvalid ),
};
```

```
int fcount = sizeof ( ffuncs ) / sizeof ( struct fnc_data ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in funclist.c, installs the functions in `ffuncs` as demand field functions:

```
sm_install ( FIELD_FUNC, ffuncs, &fcount ) ;
```

Grid Functions

You can install one or more demand grid functions that can be called explicitly at different stages of grid execution:

- Grid entry and exit.
- Grid row entry and exit.
- Grid validation.

Panther can recognize two events as grid entry: when the cursor enters a grid; and when the screen's current grid is reactivated because an overlying window closes, if setup variable `EXPHIDE_OPTION` is set to `ON_EXPHIDE`.

On grid entry, the grid's entry function is executed first, then the grid's row entry function. The grid row exit and entry functions are repeatedly called each time the cursor exits the current row and enters another one.

Panther can recognize two events as grid exit: when the cursor leaves a grid; and when a window overlays the grid's screen, if setup variable `EXPHIDE_OPTION` is set to `ON_EXPHIDE`. On exit, the grid row exit function is called before the grid exit function.

Grid Function Arguments

All grid functions receive three arguments in this order:

- An integer that contains the grid's base widget number—that is, the base widget number of the grid's leftmost array, whether or not it is hidden or the grid uses the first row as a title row.
- An integer that contains the occurrence number of the current grid row. This argument is supplied only to grid row entry or row exit functions; otherwise this argument is 0.
- An integer bitmask that indicates why the function was called.

The last parameter can have one or more flags set. The following sections describe these flags:

K_ENTRY

The function was called on grid or row entry.

Equivalent: `if (param3 & K_ENTRY)`

K_EXIT

The function was called on grid or row exit. Note that if neither `K_ENTRY` nor `K_EXIT` are set, the grid is undergoing validation.

Equivalent: `if (param3 & K_EXIT)`

K_EXPOSE

The function was called because a window overlying this grid's screen opened or closed:

`K_EXPOSE` and `K_ENTRY` are set: the overlying window closed and the grid is exposed.

`K_EXPOSE` and `K_EXIT` are set: the overlying window opened and the grid is hidden.

Equivalent: `if(param3 & K_EXPOSE)`

K_KEYS

Mask for the flags that tell which keystroke or event caused entry, exit, or validation for the grid or grid row. The intersection of this mask and the third parameter to the function should be tested for equality against one of the next six flags.

K_NORMAL

A "normal" key caused the cursor to enter or exit the grid or grid row in question. For grid or grid row entry, "normal" keys are `NL`, `TAB`, `HOME`, and `EMOH`. For grid exit, only `TAB` and `NL` are considered "normal."

Equivalent: `if((param3 & K_KEYS)==K_NORMAL)`

K_BACKTAB

The `BACKTAB` key caused the cursor to enter or exit the grid or grid row.

Equivalent: `if((param3 & K_KEYS)==K_BACKTAB)`

K_ARROW

An arrow key caused the cursor to enter or exit the grid or grid row.

Equivalent: `if((param3 & K_KEYS)==K_ARROW)`

K_SVAL

The grid is being validated as part of screen validation.

Equivalent: `if((param3 & K_KEYS)==K_SVAL)`

K_USER

The grid is being validated directly from the application with `sm_fval`.

Equivalent: `if((param3 & K_KEYS)==K_USER))`

K_OTHER

A key other than `BACKTAB`, an arrow key, or a “normal” key caused the cursor to enter or exit the grid or grid row. For entry, “normal” keys are `NL`, `TAB`, `HOME`, and `EMOH`. For exit, only `TAB` and `NL` are considered “normal.”

Equivalent: `if((param3 & K_KEYS)==K_OTHER)`

K_INSDEL

The `INSL` or `DELL` key caused grid row exit or entry. Use this flag in row exit or entry processing to determine whether the entry or exit event resulted from the user inserting or deleting a grid row. For example, the following code differentiates between row exit events that are caused by `INSL` or `DELL` inserting or deleting a gird row, and other actions that might cause row exit, such as the user pressing `TAB`:

```
if (param4 & K_EXIT)
{
    if((param3 & K_KEYS) == K_INSDEL)
   // if exit occurred because user inserted or deleted
    // row, do nothing
    {
        return
    }
    // 'real' exit occurred, so perform exit processing
    ...
}
```

Grid Function Returns

Grid functions return meaningful values only if called as the grid's validation function—`0` if successful, non-zero if not.

Installation of Demand Grid Functions

You can install multiple functions as demand grid functions. The following statements, typically found in `funclist.c`, include two all-purpose grid entry and exit functions `gridEntry` and `gridExit` in the `fnc_data` structure `grdfuncs`:

```
struct fnc_data grdfuncs[] =
{
   SM_INTFNC( "gridEntry", gridEntry ),
   SM_INTFNC( "gridExit", gridExit ),
};
```

```
int gcount = sizeof ( grdfuncs ) / sizeof (struct fnc_data) ;
```

The following line of code, typically found in the function `sm_do_uinstalls` in `funclist.c`, installs the functions in `grdfuncs` as demand grid functions:

```
sm_install ( GRID_FUNC, grdfuncs, &gcount ) ;
```

Tab Control Functions

You can install one or more demand tab card functions that can be called explicitly at different events:

- Tab card entry and exit.
- Tab card hide and expose.

Since the index tab on each card is a field, refer to refer to page 44-14 for information on field functions.

Tab Control Function Arguments

All tab card functions receive two arguments in this order:

- The object id of the tab card.
- An integer bitmask that indicates why the function was called.

The last parameter can have one or more flags set. The following sections describe these flags:

K_ENTRY

The function was called on tab card entry.

Equivalent: `if(param2 & K_ENTRY)`

K_EXIT

The function was called on tab card exit.

Equivalent: `if(param2 & K_EXIT)`

K_EXPOSE

The function was called because the screen containing this card opened or was exposed, because the overlying tab card was hidden, or because this tab card is the new topmost card.

Equivalent: `if(param2 & K_EXPOSE)`

For the card hide event, all three flags are clear.

Group Functions

Panther calls group functions on entry, exit, and validation of groups. You can install an automatic group function is invoked on entry, exit, and validation for all groups. Each group can invoke its own functions for these events through its entry, exit, and validation functions. You can specify these event functions in the group's properties window, accessed through the screen editor.

The automatic group function is executed on all group events. On entry, the automatic group function is executed before the group's entry function. On exit, the group's exit and validation functions is called first, and then the automatic group function. If the group has a JPL group module, this module is executed only after the group functions.

Two events are recognized as group entry: when the cursor enters a group; and when the screen's current group is reactivated because an overlying window closes.

Two events are recognized as group exit: when the cursor leaves a group; and when a window overlays the group's screen.

Group validation functions are called under the following conditions:

- As part of group validation, when you exit the group by pressing `TAB` or selecting from an autotab group. `BACKTAB`, arrow keys and mouse clicks outside the group also cause validation, unless the setup variable `IN_VALID` is changed from its default setting to `OK_NOVALID`.
- As part of screen validation when the user presses `XMIT`.
- When the application code calls library functions for group validation.

Note that if a group contains a widget that has its own validation function, this function is called when the widget is selected. The validation function of a check box is also called when the widget is deselected.

Notes: It is not advisable to bring up a dialog box, such as a message dialog, from a group entry function, since opening a screen between a mouse down and a mouse up event yields undefined results.

Group Function Arguments

All group functions receive two arguments:

- A pointer to a null-terminated character string that contains the group's name.
- An integer bitmask that indicates whether the group has been validated and why the function was called.

The flags that can be set on a group's bitmask are the same as for a single widget. For a description of these flags, refer to page 44-16.

Group functions are called for validation whether or not the group has already been validated. You can test the `VALIDED` and `MDT` bits to avoid redundant processing.

Group Function Returns

Group functions called on entry or exit should return 0. Group functions called for validation should return 0 if the group selections pass the validation criteria. A non-zero return code should indicate that the group failed validation. If the return value is 1, the cursor position remains unchanged. Any other non-zero return value repositions the cursor to the group that failed validation.

Installation of an Automatic Group Function

You can install only one function as the automatic group function. The following statement, usually found in funclist.c, includes the automatic group function `auto_gfunc` in the fnc_data structure `autogrp_struct`. To see the code for this function, refer to page 44-71.

```
struct fnc_data autogrp_struct = SM_OLDFNC( 0, auto_gfunc ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, installs `auto_gfunc` as the default group function:

```
sm_install ( DFLT_GROUP_FUNC, &autogrp_struct, (int *) 0 ) ;
```

Installation of Demand Group Functions

You can install multiple functions as demand group functions. The following statements, usually found in `funclist.c`, include two all-purpose group entry and exit functions `gEntry` and `gExit` in the `fnc_data` structure `gfuncs`:

```
struct fnc_data gfuncs[] =
{
   SM_OLDFNC( "gEntry", gEntry ),
   SM_OLDFNC( "gExit", gExit ),
};
```

```
int gcount = sizeof ( gfuncs ) / sizeof ( struct fnc_data ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, installs the functions in `gfuncs` as demand widget functions:

```
sm_install ( GROUP_FUNC, gfuncs, &gcount ) ;
```

Client Authentication Functions

With the JetNet/Tuxedo middleware adapters, a client authentication function is used by the middleware's authentication server during client initialization. This function, specified by `client_init`'s `DATAFUNC` option, provides the data required to authorize the client connection. For more information about client authentication options in JetNet and Tuxedo, refer to page 2-11 in the *Programming Guide*.

Client Authentication Arguments

A client connection authentication (`DATAFUNC`) function requires five arguments:

- A pointer to the address where the client authentication data is located. This address is set by the function and must remain valid on return. The type of the user data contained at the address depends on the authentication server.
- A string containing the argument passed to `client_init` for the `USER` option, if used; otherwise `NULL`. This value must not be changed.
- A string containing the argument passed to `client_init` for the `CLIENT` option, if used; otherwise `NULL`. This value must not be changed.
- A string containing the argument passed to `client_init` for the `PASSWORD` option, if used; otherwise `NULL`. This value must not be changed.
- A string containing the argument passed to `client_init` for the `GROUP` option, if used; otherwise `NULL`. This value must not be changed.

Client Authentication Returns

`DATAFUNC` functions should return a long containing the non-negative length of the authentication data at the address (first argument). A positive value must be accompanied by a non-zero address in the authentication data address. If the function returns a negative value, or if the address is `NULL` when a positive value is returned, `client_init` raises a `TP_DATAFUNC_FAILED` exception.

Installation

You can install multiple `DATAFUNC` functions. The following statements, typically found in `funclist.c`, includes the `DATAFUNC` function `user_datafunc` in the `fnc_data` structure `authpre`. Examples are given for both a single `DATAFUNC` and multiple functions.

Install a single `DATAFUNC` function as follows:

```
struct fnc_data authpre =

   SM_STRFNC ("user_datafunc", user_datafunc);
```

The following line of code, typically found in the function `sm_do_uinstalls` in `funclist.c`, installs `user_datafunc` as the default client authentication connection function:

```
sm_install ( TP_INITDATA_FUNC, authpre, (int *)0);
```

You cam install multiple `DATAFUNC` functions by defining them in an array of `fnc_data` structures:

```
static struct fnc_data authpre[] =
{
   SM_STRFNC ("user_datafunc1", user_datafunc1),
   SM_STRFNC ("user_datafunc2", user_datafunc2)
};

static int acount =

      sizeof(authpre) / sizeof (struct fnc_data);
```

The following line of code, typically found in the function `sm_do_uinstalls` in `funclist.c`, installs the functions in `authpre` as `DATAFUNC` functions, to be called depending on the argument supplied by `client_init's` `DATAFUNC` option:

```
sm_install (TP_INITDATA_FUNC, authpre, &acount);
```

Client Post-Connection Functions

With the JetNet/Tuxedo middleware adapters, `client_init` can include a `POSTFUNC` option that specifies a function to be paired with its `DATAFUNC` function. A `POSTFUNC` function is always called whether or not the client authentication is successful. For more information about client authentication options in JetNet and Tuxedo, refer to page 2-13 in the *Programming Guide*.

Client Post-Connection Arguments

All `POSTFUNC` functions require six arguments:

- The address of the authentication data, as obtained from the prior call to the `DATAFUNC` function.
- A long containing the length of the authentication data, as obtained by the return value from the `DATAFUNC` function.
- A string containing the argument passed to `client_init` for the `USER` option, if used; otherwise `NULL`. This value must not be changed.
- A string containing the argument passed to `client_init` for the `CLIENT` option, if used; otherwise `NULL`. This value must not be changed.
- A string containing the argument passed to `client_init` for the `PASSWORD` option, if used; otherwise `NULL`. This value must not be changed.
- A string containing the argument passed to `client_init` for the `GROUP` option, if used; otherwise `NULL`. This value must not be changed.

Client Post-Connection Returns

```
void (none)
```

Installation

You can install multiple `POSTFUNC` functions. The following statements, typically found in `funclist.c`, includes the `POSTFUNC` function `user_postfunc` in the `fnc_data` structure `authpost`. Installation instructions are shown for both a single `POSTFUNC` and multiple functions.

Install a single `POSTFUNC` function as follows:

Define the function in a `fnc_data` structure.

```
struct fnc_data authpost =

   SM_STRFNC ("user_postfunc", user_postfunc);
```

The following line of code, typically found in the function `sm_do_uinstalls` in `funclist.c`, installs `user_postfunc` as the default client authentication post-connection function:

```
sm_install ( TP_INITPOST_FUNC, authpost, (int *)0 );
```

You can install multiple `POSTFUNC` functions by defining them in an array of `fnc_data` structures:

```
static struct fnc_data authpost[] =
{
   SM_STRFNC ("user_postfunc1", user_postfunc1),
   SM_STRFNC ("user_postfunc2", user_postfunc2)
};

static int acount =

       sizeof(authpost) / sizeof (struct fnc_data);
```

The following line of code, typically found in the function `sm_do_uinstalls` in funclist.c, installs the functions in `authpost` as client authentication post-connection functions, to be called depending on the argument to the `POSTFUNC` option used in a call to the `client_init` command.

```
sm_install (TP_INITPOST_FUNC, authpost, &acount);
```

For further information on client authentication, refer to `client_init`.

Help Function

A help function installs a driver to invoke an external help facility such as `WINHELP` from your application. This driver gets a single argument from its caller, which contains a help context identifier. It is the responsibility of the help driver to pass this identifier to the help facility.

Help Function Arguments

The help function gets a single string that contains the help context identifier.

Help Function Returns

```
PI_ERR_NONE (success) or
PI_ERR_NO_MORE (failure).
```

Installation

You can install only one function as a help function. The following statement, usually found in `funclist.c`, include the help function `sm_PiXmDynaHook` in the `fnc_data` structure `hlp_struct`. To see the code for this function, refer to page 44-73.

```
struct fnc_data hlp_struct = SM_OLDFNC( 0, sm_PiXmDynaHook );
```

The following line of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, installs `sm_PiXmDynaHook` as the default help function:

```
sm_install ( EXTERNAL_HELP_FUNC, &hlp_struct, (int *) 0 );
```

Timeout Functions

Panther periodically calls the installed timeout functions while the keyboard input function awaits user input. You can use timeout functions to poll or otherwise manipulate communications resources, or to update the screen display. You can install multiple timeout functions with different time lapse specifications, measured in minutes, seconds, or tenths of seconds. Each timeout function is called when its timeout interval elapses.

Timeout functions are called from the lowest level of keyboard or mouse input. When they are installed, the device driver clock on the terminal input device is set to time out on its character read operation. If Panther does not read any character in the time interval specified by a timeout function, it calls that function before it tries to read another character.

Timeout Function Arguments

Timeout functions get one integer argument that tells why the function was called:

TF_TIMEOUT

No keyboard activity occurred for the amount of time specified by this function's timeout interval.

TF_RESTART

Keyboard input was received during execution of the timeout function.

Timeout Function Returns

A timeout function should return a code that indicates whether Panther should keep calling the timeout function after each lapse of the timeout interval:

TF_KEEP_CALLING

Keep calling the user function each timeout the interval elapses.

TF_STOP_CALLING

Do not call the timeout function again until keyboard input is received.

Installation

You can install multiple timeout functions. The following statements, usually found in `funclist.c`, installs a single timeout function `screen_saver` in the `fnc_data` structure `timeout_funcs`. To see the code for this function, refer to page 44-78. The first member of this structure specifies units of measurement: `TF_TENTHS` (tenths of seconds), `TF_SECONDS`, or `TF_MINUTES`. The fifth member specifies the timeout interval as a multiple of these units.

```
struct fnc_data timeout_funcs[] =
{
   { TF_MINUTES, screen_saver, 0, 0, 10, 0 }
};

int tcount = sizeof ( timeout_funcs )
                    / sizeof (struct fnc_data ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, installs the function in `timeout_funcs` as a timeout function:

```
sm_install ( TIMEOUT_FUNC, timeout_funcs, &tcount ) ;
```

Timer Functions

Panther periodically calls installed timer functions while the keyboard input function awaits user input. Timer functions differ from timeout functions in that the interval for timer functions is not reset after user input. You can use timer functions to poll or otherwise manipulate communications resources, or to update the screen display. You can install multiple timer functions with different time lapse specifications, measured in minutes, seconds, or tenths of seconds. Each timer function is called when its interval elapses.

Timer Function Arguments

Timer functions get one integer argument that tells why the function was called:

TF_TIMEOUT

The timer's interval has expired.

TF_RESTART

There was a timer interval expiration for which the function was not called. This could happen, for example, during a long database operation. `TF_RESTART` calls only occur after `TF_TIMEOUT` calls. If the timer function returns `TF_STOP_CALLING`, no more `TF_RESTART` calls will occur before the next `TF_TIMEOUT` call.

Timer Function Returns

A timer function should return a code that indicates whether Panther should keep calling the function after each lapse of the timer interval:

TF_KEEP_CALLING

Call the timer function extra times if timer expiration calls were missed.

TF_STOP_CALLING

Do not call the timer function again until the next timer expiration.

Installation

You can install multiple timer functions. The following statements installs a single timer function `poll_database` in the `fnc_data` structure `timer_funcs`. The first member of this structure specifies units of measurement: `TF_TENTHS` (tenths of seconds), `TF_SECONDS`, or `TF_MINUTES`. The fifth member specifies the timer interval as a multiple of these units.

```
struct fnc_data timer_funcs[] =
{
   { TF_MINUTES, poll_database, 0, 0, 10, 0 }
};
```

```
int tcount = sizeof ( timer_funcs )
                   / sizeof (struct fnc_data ) ;
```

The following line of code installs the function in `timer_funcs` as a timeout function:

```
sm_install ( TIMER_FUNC, timer_funcs, &tcount ) ;
```

Key Change Function

A key change function can be called whenever Panther reads a key from the keyboard. You can use key change functions to intercept, process, or translate keystrokes at the logical key level. Key change functions can be useful alternatives to using `sm_keyoption`.

Panther calls the key change function once for each key that it gets from the keyboard or the playback function.

Note: The key change function ignores any keys placed on the input queue by `sm_ungetkey` *or* `jm_keys`.

Key Change Function Arguments

A key change function gets one integer argument, the Panther logical key that is read from the keyboard or received from a playback function.

Note: The key change function is not called for the following keys: `MNBR`, `ALSYS`, and `ALT` keys.

Key Change Function Returns

A key change function returns the key to be input into the application by `sm_getkey`. If the key change function returns 0, `sm_getkey` gets the next key from the keyboard.

Installation

You can install only one key change function. The following statement, usually found in `funclist.c`, includes key change function `keychg` in the `fnc_data` structure `keychg_struct`. To see the code for this function, refer to page 44-79.

```
struct fnc_data keychg_struct = SM_OLDFNC( 0, keychg ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, installs `keychg` as the key change function:

```
sm_install ( KEYCHG_FUNC, &keychg_struct, (int *) 0 ) ;
```

Error Function

Panther calls the installed error function when it issues an error message—invoked either by a Panther error or by a call to one of Panther's error message functions—for example, `sm_fquiet_err` or `sm_ferr_reset`. You can use the error function for special error handling—for example, to write all error messages to a log file.

Error Function Arguments

The error function gets three arguments in this order:

- The number of the message to display—for Panther messages, as defined in `smerror.h`; for user-defined messages, as defined in user-created message header files. If the calling function passes a text string to display, this argument is -1.
- The text of the message to display. If the calling function passes a message number, this argument is 0.
- Tells whether to display the message in quiet mode: a value of `1` specifies yes, a value of `0` specifies no. The value will be `1` for messages displayed by JPL calls to `msg qui_msg` and `msg quiet` and the message functions `sm_fquiet_err`, `sm_fqui_msg`, `sm_quiet_err` and `sm_qui_msg`.

Error Function Returns

If the error function returns 0 to its caller, the calling message function continues processing. If this function returns a non-zero value, the calling message function returns immediately.

Installation

You can install only one error function. The following statement, usually found in `funclist.c`, includes error function `myerr` in the `fnc_data` structure `err_struct`.

```
struct fnc_data err_struct = SM_OLDFNC( 0, myerr ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, installs `myerr` as the error function. To see the code for this function, refer to page 44-81 .

```
sm_install ( ERROR_FUNC, &err_struct, (int *) 0 ) ;
```

Insert Toggle Function

An insert toggle function is called when the data entry mode switches between insert and overstrike mode—for example, when the user chooses Insert. You can use this function to display a message that indicates the current mode.

Panther automatically installs an insert toggle function that changes the cursor style when the mode is changed. If an application has its own insert toggle function installed, Panther uninstalls its insert toggle function; the insert toggle function that you install can call Panther's insert toggle function directly.

Arguments

This function gets one integer argument, which specifies the new mode:

- `1`—Insert mode
- `0`—Overstrike mode

Returns

The insert toggle function should return 0.

Installation

You can install only one insert toggle function. The following statement, usually found in `funclist.c`, includes the insert toggle function `inscrsr` in the `fnc_data` structure `keychg_struct`. To see the code for this function, refer to page 44-81.

```
struct fnc_data instgl_struct = SM_OLDFNC( 0, inscrsr ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, installs `inscrsr` as the insert toggle function:

```
sm_install ( INSCRSR_FUNC, &instgl_struct, (int *) 0 ) ;
```

Check Digit Function

A check digit function is called during validation of any widget whose Check Digit property is set. You use a check digit function to perform your own check digit algorithm. If no check digit function is installed, the library function `sm_ckdigit` is used, whose source is distributed in source form.

Because `sm_ckdigit` source is available, you can implement your own algorithm by directly modifying this library function and linking it to your application. However, if your linker does not let you override library functions, you must install your own check digit function.

Arguments

The check digit function gets these arguments:

- A pointer to a null-terminated string that contains the widget's contents.
- The occurrence number for the current widget.
- The modulus as specified in the Check Digit property.
- The minimum number of digits as specified in the Minimum Digits property.

Returns

The check digit function should return 0 if the widget passes check digit validation. If the function returns a non-zero value, the cursor is repositioned to the offending widget and the widget is not marked as validated.

Installation

You can install only one check digit function. The following statement, usually found in `funclist.c`, includes the check digit function `ckdigit` in the `fnc_data` structure `ckdgt_struct`.

```
`struct fnc_data ckdgt_struct = SM_OLDFNC( 0, ckdigit ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, installs `ckdgt` as the check digit function:

```
sm_install ( CKDIGIT_FUNC, &ckdgt_struct, (int *) 0 ) ;
```

Initialization and Reset Functions

The initialization and reset functions are called on display setup and reset, respectively. You can use the initialization function to set the terminal type, and the reset function to handle any cleanup that the application requires on exit.

The initialization function is called from the library function `sm_initcrt`. It is called before Panther allocates its own memory structures or sets the physical display. Unlike other installed functions, the initialization function should be installed before `sm_initcrt` is called. Consequently, you cannot place the installation code for this function in the funclist.c function `sm_do_uinstalls`.

The reset function is called from the library function `sm_resetcrt` after Panther releases its memory and resets the physical display. Because Panther's abort function `sm_cancel` calls `sm_resetcrt` before the application terminates, it calls the reset function at application exit whether the exit is graceful or not.

You might need to set interrupt handlers to ensure that `sm_cancel` is called at all the necessary hardware and software interrupt signals. You should set these either in the `funclist.c` function `sm_do_uinstalls`, or in the function installed as an initialization function.

Arguments

The initialization function is passed a single argument, a 30-byte character buffer that contains a null-terminated string mnemonic for the terminal type in use. This is mainly used for operating systems without an environment. You can use this function to get the terminal type in some system-specific way.

The reset function is passed no arguments.

Returns

Both the initialization and reset functions should return 0.

Installation

You can install only one initialization and one reset function. Initialization functions are called by `sm_initcrt` and so must be installed in `jmain.c` before the call to `sm_initcrt`:

```
struct fnc_data uninit_struct = SM_OLDFNC( 0, uinit ) ;
```

```
sm_install ( UINIT_FUNC, &uinit_struct, (int *) 0 ) ;
```

The reset function can be installed like other functions in `funclist.c`. This function is called from `sm_resetcrt`, and is consequently called even if the application terminates abnormally:

```
struct fnc_data ureset_struct = SM_OLDFNC( 0, ureset ) ;
```

```
sm_install ( URESET_FUNC, &ureset_struct, (int *) 0 ) ;
```

To see sample initialization and reset functions, refer to page 44-82.

Record and Playback Functions

Panther provides hooks for recording and playing back keystrokes. You can use this facility to create simple macros, or to perform regression testing on a Panther application. Be careful that record and playback functions are not in use simultaneously:

- `sm_getkey` calls the record function just before it returns a translated key value to the application.
- `sm_getkey` also calls the playback function in place of a read from the keyboard.

Characters are recorded after the key change function processes them, but are played back before key change translation; consequently, some key change functions might prevent accurate playback of recorded keystrokes. Refer to the description of `sm_getkey` for more information.

Accurate regression testing might require the playback function to pause and flush the output, in order to simulate a realistic rate of typing, and to call a timeout function.

Arguments

The record function gets a single integer argument, the Panther logical key to record. This key is usually recorded in some fashion for later playback.

The playback function gets no arguments.

Returns

The record function should return 0. The playback function should return the logical key previously recorded.

Installation

You can install only one record and one playback function. The following statements, usually found in funclist.c, include the record and playback functions record and play in the `fnc_data` structures `record_struct` and `play_struct`, respectively. To see the code for these functions, refer to page 44-84.

```
struct fnc_data record_struct = SM_OLDFNC( 0, record ) ;
```

```
struct fnc_data play_struct = SM_OLDFNC( 0, play ) ;
```

The following lines of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, install record and play as the record and playback functions:

```
sm_install ( RECORD_FUNC, &record_struct, (int *) 0 ) ;
```

```
sm_install ( PLAY_FUNC, &play_struct, (int *) 0 ) ;
```

Control Functions

Control functions are called either through control strings or by the call command in a JPL procedure. Because control functions take only one argument—a pointer to a copy of the control string that invoked it—you can install as control functions those functions whose argument list is especially long or complex—for example, a SQL statement.

All control functions are demand types—that is, they must be explicitly named by one of the aforementioned callers.

Arguments

A control function receives one argument, a pointer to a copy of the control string or call command that invoked it. This string is stripped of its leading caret ^ or call verb. Panther identifies only the first word of the control string as the function name; the rest of the string can be parsed and used as arguments by the function.

Returns

Control functions can return any integer. You can use the return value for conditional control branching in a control string's target lists. If the function returns a function key that is not a value in the target list, Panther processes the key and executes its control string, if any.

Installation

You can install multiple functions as control functions. The following statements, typically found in `funclist.c`, include two control functions `mark_low` and `mark_high` in the `fnc_data` structure `mark_funcs`. To see the code for these functions, refer to page 44-87.

```
struct fnc_data mark_funcs[] =
{
   SM_OLDFNC( "mark_low", mark_low ),
   SM_OLDFNC( "mark_high", mark_high ),
};

int markcount = sizeof ( mark_funcs )
                        / sizeof ( struct fnc_data ) ;
```

The following line of code, typically found in the function `sm_do_uinstalls` in `funclist.c`, installs the functions in `mark_funcs` as control functions:

```
sm_install ( CONTROL_FUNC, mark_funcs, &markcount ) ;
```

Status Line Function

Panther calls the status line function just before the status line is flushed or physically written to the terminal display. Because of delayed write, this might not coincide with calls to functions that specify message line text. You typically use this function for terminals that require special status line processing.

Arguments

The status line function gets no arguments. It can access copies of the text and attributes about to be flushed to the status line through the following calls to library functions:

```
stat_attr = sm_pinquire(SP_STATLINE);
```

```
stat_attr = sm_pinquire(status, SP_STATATTR);
```

Note that in the case of the status text and status attribute globals, `sm_pinquire` returns a pointer to a temporary copy of the arrays. You should copy these to a save location before using them.

Returns

If the status line function returns 0, Panther continues its usual processing and writes out the status line. If the function returns a non-zero value, Panther assumes that the function handles the physical write of the status line.

Installation

You can install only one status line function. The following statement, usually found in `funclist.c`, includes the status line function `statln` in the `fnc_data` structure `stat_struct`. To see the code for this function, refer to page 44-96.

```
struct fnc_data stat_struct = SM_OLDFNC( 0, statln ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, installs `statln` as the status line function:

```
sm_install ( STAT_FUNC, &stat_struct, (int *) 0 ) ;
```

Video Processing Function

A character-based application can use the video processing function for special handling of various video sequences. GUI applications ignore the video processing function. Use your own video processing function only if Panther has no video file that supports a specific terminal type. Panther's output function calls the video processing function just before it displays data on a Panther screen; consequently, this function should perform only low-level processing.

Video processing functions should not call Panther library functions.

Arguments

The video processing function receives two arguments:

- An integer video processing code defined in the header file `smvideo.h` and outlined in Table 44-2.
- A pointer to an array of integers with parameters for the video processing code. The number of parameters passed depends on the operation as shown in Table 44-2. For video processing codes that require no arguments, supply `NULL`.

Table 44-2 Video processing codes

Code	Parameters	Action
`V_ARGR`		Remove area attribute.
`V_ASGR`	11	Set area graphics rendition.
`V_BELL`		Visible alarm sequence.
`V_CMSG`		Close message line.
`V_COF`		Turn cursor off.
`V_CON`		Turn cursor on.

Table 44-2 Video processing codes

Code	Parameters	Action
`V_CUB`	1	Cursor back (left).
`V_CUD`	1	Cursor down.
`V_CUF`	1	Cursor forward (right).
`V_CUP`	2	Set cursor position (absolute).
`V_CUU`	1	Cursor up.
`V_ED`		Erase entire display.
`V_EL`		Erase to end of line.
`V_EW`	5	Erase window to background.
`V_INIT`		Initialization string.
`V_INSON`		Set insert cursor style.
`V_INSOFF`		Set overstrike cursor style.
`V_MODE0`		Set graphics mode (also `V_MODE1`,2,3).
`V_MODE4`		Single character graphics mode (also `V_MODE5`,6).
`V_OMSG`		Open message line.
`V_RCP`		Restore cursor position.
`V_REPT`	2	Repeat character sequence.
`V_RESET`		Reset string.
`V_SCP`		Save cursor position.
`V_SGR`	11	Set latch graphics rendition.

Returns

If the function returns 0, normal processing is continued. If it returns a non-zero value is returned, Panther assumes that the function handled the operation. This lets you implement only necessary operations.

Installation

You can install only one video processing function. The following statement, usually found in `funclist.c`, includes the video processing function video in the `fnc_data` structure `video_struct`.

```
struct fnc_data video_struct = SM_OLDFNC( 0, video ) ;
```

The following line of code, usually found in the function `sm_do_uinstalls` in `funclist.c`, installs video as the video processing function:

```
sm_install ( VPROC_FUNC, &video_struct, (int *) 0 ) ;
```

Database Driver Hook Functions

Panther's database drivers have three hook functions that can be used to write database error handlers: `ONERROR`, `ONENTRY`, and `ONEXIT`. For more information about using these commands, refer to page 11-42 in the *Programming Guide*.

Transaction Manager Event Functions

When the transaction manager traverses the transaction tree in order to issue commands to each table view or server view, it checks to see if any of these table or server views have a function property specified. If so, the transaction manager looks for it among the installed functions and calls it. If the function contains processing for the current transaction event, that processing is completed.

If the return code is `TM_PROCEED`, the transaction manager then calls the database-specific and the common models for the same transaction event.

If the return code is `TM_OK`, the transaction manager continues to the next table view or the next transaction event. If the return code is `TM_CHECK`, `TM_CHECK_ONE_ROW`, or `TM_CHECK_SOME_ROWS`, the transaction manager pushes an event onto the stack to check for database errors.

For information about writing transaction event functions, refer to page 32-1.

Arguments

Transaction manager functions are passed a single integer argument that corresponds to the transaction event. Transaction events and their integer values are listed in `tmusubs.h`.

Returns

Table 44-3 summarizes possible return codes for transaction manager functions:

Table 44-3 Return codes for transaction manager functions

Return value	Description
`TM_OK`	Event processing succeeded.

Table 44-3 Return codes for transaction manager functions

Return value	Description
`TM_FAILURE`	Event processing failed.
`TM_PROCEED`	After completing the function, proceed to call the transaction model for this event, as if this function had never been called.
`TM_CHECK`	Test to see if an error occurred. This is used in data base-specific transaction models to check for SQL execution errors.
`TM_CHECK_ONE_ROW`	In addition to an error test, test that exactly one row was affected by the processing.
`TM_CHECK_SOME_ROWS`	In addition to an error test, test that one or more rows were affected by the processing.
`TM_UNSUPPORTED`	Event was not recognized.

Installation

You can specify a transaction manager function for each table view or server view in a screen for insert, update, or delete processing. With the table view selected, enter the name of the function in the Function property (under Transaction). If the function affects the SQL generation of `SELECT` statements, the function must be entered on the appropriate server view.

The function itself can be either a JPL procedure or C function that is installed in the prototyped function list.

Errors

When a screen is opened, the transaction manager reports an error if the function cannot be found. As a result of this error, the transaction manager does not start its transaction.

Sample Functions

The following sections show sample code for commonly used function types.

Prototyped

This section has two sample functions:

- `mark_flds` gets a range of values and highlights all widgets whose data is within that range.

  report generates a report whose type and output device vary according to the supplied arguments.

- `mark_flds` gets a range of values and highlights all widgets whose data is within that range. This function takes two integer arguments which specify the low and high ends of the range. If the first argument is less than the second, all widgets on the screen with numeric values between the two arguments are temporarily highlighted. If the first argument is greater than the second, all widgets on the screen with numeric values that are not between the two widgets are highlighted.

For example, this control string highlights all values on the screen between zero and 500:

```
^mark_flds (0, 500)
```

The next control string highlights all values on the screen that are greater than 1000 or less than -300:

```
^mark_flds (1000, -300)
```

The following code comprises the entire `mark_flds` function.

```
/* Include Files */
#include "smdefs.h"   /* screen manager Header File */
#include "smglobs.h"  /* screen manager Globals */
```

```
/* Macro Definitions... */
/* Attributes used to mark fields */

#define MARK_ATTR REVERSE | HILIGHT | BLINK

int
mark_flds ( bound1, bound2 )

int bound1 ; /* First Boundary on fields to mark */
int bound2 ; /* Second Boundary on fields to mark */

{
   int fld_num ;    /* Field Number */
   char *fld_data;  /* Field Data */
   double fld_val ; /* Field Value */
   int num_of_flds ;/* Number of Fields */
   int *old_attrib ;/* Array of old attributes */

   /* Determine number of fields */

   num_of_flds = sm_inquire ( SC_NFLDS ) ;

   /* Allocate memory for attribute array */

   old_attrib = (int *)calloc ( num_of_flds,
       sizeof ( int ) ) ;

   /* Cycle through all the fields on the screen */

   for ( fld_num = 1 ; fld_num <= num_of_flds ; fld_num++ )
   {
      /* Store away old attributes */

      old_attrib[fld_num-1] =
          sm_finquire ( fld_num, FD_ATTR ) ;

      /* Make sure it is a field with numbers */

      fld_data = sm_strip_amt_ptr ( fld_num, NULL ) ;
      if ( ! *fld_data ) continue ;

      /* Create a double from it */

      fld_val = sm_dblval( fld_num ) ;

      /* See if fld_val is in bounds */

      if ( bound1 <= bound2 )
      {
         /* Mark fields between bounds. */
```

```
            if ( ( fld_val >= ( double )bound1 ) &&
                ( fld_val <= ( double )bound2 ) )
            {
               sm_chg_attr ( fld_num,
                   MARK_ATTR ) ;
            }
         }
         else
         {

            /* Mark fields outside bounds. */

            if ( ( fld_val >= ( double )bound1 ) ||
                ( fld_val <= ( double )bound2 ) )
            {
               sm_chg_attr ( fld_num,  MARK_ATTR ) ;
            }
         }
      }

      /* Wait for acknowledgement */

      sm_ferr_reset ( "Hit <space> to continue") ;

      /* Cycle again through all the fields on the screen */

      for ( fld_num = 1 ; fld_num <= num_of_flds ; fld_num++ )
      {
         /* Reset field attributes */

         sm_chg_attr ( fld_num,
             old_attrib[ fld_num - 1 ] ) ;

      }

      /* Release memory */

      free ( (char *)old_attrib ) ;
      return ( 0 ) ;
   }
```

Example 2

`report` generates a report whose type and output device vary according to the supplied arguments. This function takes two string arguments:

- The first argument specifies the report type with one of these values: `field`, `screen`, `wstack`, or `term`.

■ The second argument specifies where to output the report. If you supply a null string, the requested report is shown in a message window. For example, the following control string causes a widget report to pop up in a message window:

```
^report(“field”, “”)
```

If the second argument starts with an exclamation point (!), the remainder is interpreted as an operating system command. The report is created in a temporary file, and the name of the file is passed as an argument to the operating system command. If a tilde (~) is embedded in the command, the name of the temporary file is substituted for the tilde, otherwise the name is just appended at the end. These two control strings both cause a screen report to print on a UNIX system:

```
^report (“screen”,"!lp -c -s")
```

```
^report ("screen", "!date | cat - ~ | lp -s")
```

If the second argument starts with a vertical bar (|), the remainder is also interpreted as an operating system command. In this case, however, the report is piped into the standard input of that command. This control string prints out the last twenty lines of a window stack report on a UNIX system:

```
^report ("wstack", "| tail | lp -s")
```

Finally, if the second argument is a valid file name, the report is appended to the named file. This control string causes a display terminal report to be appended to the file `report.fil`:

```
^report("term", "report.fil")
```

The following code comprises the entire report function.

```
/* Include Files */
#include "smdefs.h"   /* screen manager Header File */
#include "smglobs.h"  /* screen manager Globals */

int

report ( report_type, report_out )
char *report_type ;     /* Type of report: field, screen,
                           wstack, or term. */
char *report_out ;      /* Output designation. */

{
   char *fn = NULL ;    /* Name of output file */
   char *ptr, *ptr1 ;   /* Character pointers */
   char msg_buf[ 128 ]; /* Message buffer */
   FILE *fp ;              /* File pointer for output */
```

```
int size ;               /* Size of output file */
int cur_no ;          /* Current field number */
int select ;          /* Current window stack index */

/* If an output designation was made... */

if ( report_out && *report_out )
{
   /* Based on what output type we designated: */

   switch ( *report_out )

   {
   case '!' :
      /* OS command. Open temp file */
      fn = tempnam ( NULL, "rprt" ) ;
      fp = fopen ( fn, "w" ) ;
      break ;

    case '|' :

      /* Pipe. Open the pipe */

      fp = popen ( report_out + 1, "w" ) ;
      break ;

   default :

      /* Other. Open the file */

      fp = fopen ( report_out, "a+" ) ;
      break ;

   }

   /* If we could not open the file, show error */

   if ( ! fp )
   {
      sprintf ( msg_buf,
          "Cannot open stream for %s.",
          report_out ) ;
      sm_ferr_reset ( msg_buf ) ;
      return ( -1 ) ;
   }
}

/* If no report output specified, open temp file for
   storing message window stuff. */

else
{
   fn = tempnam ( NULL, "rprt" ) ;
   fp = fopen ( fn, "w+" ) ;
   report_out = "" ;
}

fprintf ( fp, "  REPORT TYPE: %s ", report_type ) ;

/* Now, based on the report_type, which is the name
   with which the function was invoked, create
   the reports. Note that all newlines are
   preceded with spaces, this is so that in the
   case of the message windows we can replace
   all space-newlines with %N, the newline
   indicator for Panther windows. */

switch ( *report_type )
{
case 'F':
case 'f':
   /* Output a field report */

   fprintf ( fp, " Field Report: " ) ;

   /* Field Identifier and contents */

   cur_no = sm_getcurno ( );
   fprintf ( fp, "FIELD: %d (%s[%d]) = %s ",
       cur_no,
       sm_name ( cur_no ),
       sm_occur_no ( ),
       sm_fptr ( cur_no ) ) ;

   /* Field sizes */

   size = sm_finquire ( cur_no, FD_LENG ) ;
   fprintf ( fp, "LENGTH: onscreen: %d "
       "Max: %d ",
       size, sm_finquire ( cur_no,
       FD_SHLENG )
       + size ) ;

   fprintf ( fp, "# OCCURRENCES: onscreen: %d "
       "Max: %d ",
       sm_finquire ( cur_no, FD_ASIZE ),
       sm_max_occur ( cur_no ) ) ;

   break;

case 'S':
case 's':
   /* Output screen report */

   fprintf ( fp, "  Screen Report: " ) ;

   /* Screen Name */

   fprintf ( fp, "SCREEN: %s ",
       sm_pinquire ( SP_NAME ) ) ;

   /* How much of screen is visible */

   fprintf ( fp, "%% VISIBLE IN VIEWPORT: %d ",
       100 *
       ( sm_inquire ( SC_VNLINE ) *
       sm_inquire ( SC_VNCOLM ) ) /
       ( sm_inquire ( SC_NCOLM ) *
       sm_inquire ( SC_NLINE ) ) ) ;

   break ;

case 'w':
case 'W':

   /* Output Window stack report */

   fprintf ( fp, "  Window Stack Report: " ) ;

   /* Cycle through all the windows. */

   for ( select = 0 ;
       sm_wselect ( select ) == select ;
       select++ )
   {
      /* Window number... */

      fprintf ( fp, " Window %d: ",
          select ) ;

      /* Screen name */

      fprintf ( fp, "Screen: %s ",
          sm_pinquire ( SP_NAME ) ) ;

      /* Number of fields and groups */

      fprintf ( fp, "# of Fields: %d "
          "# of Groups: %d ",
          sm_inquire ( SC_NFLDS ),
          sm_inquire ( SC_NGRPS ) ) ;
      sm_wdeselect ( ) ;
   }
   sm_wdeselect ( ) ;
   break ;

case 'T':

case 't':

   /* Output display terminal report */

   fprintf ( fp, "  Terminal Report: " ) ;

   /* Terminal Type */

   fprintf ( fp, "TERM TYPE: %s ",
       sm_pinquire ( P_TERM ) ) ;

   /* Display mode */

   if ( sm_inquire ( I_NODISP ) )
      fprintf ( fp, "DISPLAY OFF " ) ;
   else
      fprintf ( fp, "DISPLAY ON " ) ;

   /* Input mode */

   if ( sm_inquire ( I_INSMODE ) )
      fprintf ( fp, "INSERT MODE " ) ;

   else
      fprintf ( fp, "TYPEOVER MODE " ) ;

   /* Block mode */

   if ( sm_inquire ( I_BLKFLGS ) )
      fprintf ( fp, "BLOCK MODE " ) ;

   /* Physical display size */

   fprintf ( fp, "DISPLAY SIZE: %d x %d ",
       sm_inquire ( I_MXLINES ),
       sm_inquire ( I_MXCOLMS ) ) ;
   break;

default:

   /* Unrecognized report type */

   fprintf ( fp, "Illegal Report Type  " ) ;
   return ( -3 ) ;
}

/* Once again, based on the type output... */
```

```
switch ( *report_out )
{
case '|' :

   /* It was a pipe, so close it. */

   pclose ( fp ) ;
   sm_ferr_reset ( "Pipe successful" ) ;
   break ;

case '!' :

   /* It was an O/S command. Close file... */

   fclose ( fp ) ;

    /* Gobble up the exclamation point */

   report_out++;

   /* Look for tildes */

   if ( ptr = strchr ( report_out, '~' ) )
   {
      /* Found the tilde. Substitute the
         file name for it. */

      *ptr = '';
      sprintf ( msg_buf, "%s%s%s",
          report_out, fn, ptr+1 ) ;
   }

   else

   {
      /* No tilde. Append file name to
            O/S command. */

      sprintf ( msg_buf, "%s %s",
          report_out, fn ) ;
   }

   /* Do the command. */

   system ( msg_buf ) ;

   /* Delete temp file and free its name. */

   remove ( fn ) ;
   free ( fn ) ;
   sm_ferr_reset ( "Command Invoked" ) ;
   break ;
```

```
case '':

   /* Message window. Get size of file... */

   size = ftell ( fp ) ;

   /* Allocate memory for it. */

   ptr = malloc ( size + 1 ) ;

   /* Rewind the file */

   fseek ( fp, SEEK_SET, 0 ) ;

   /* Read it into the malloced buffer. */

   fread ( ptr, sizeof ( char ), size, fp ) ;

   /* Close and delete file, free file name */

   fclose ( fp ) ;
   remove ( fn ) ;
   free ( fn ) ;

   /* null terminate memory buffer of report */

   ptr[size] = '';

   /* Replace all space-newlines with %N */

   for ( ptr1 = ptr ;
       ptr1 = strchr ( ptr1, '' ) ;
       ptr1++ )
   {
      ptr1[-1]='%';
      ptr1[0]='N';
   }

   /* Pop up the message window */

   sm_message_box
      ( ptr, 0, SM_MB_OK|SM_MB_ICONNONE, 0) ;

   /* Free up the malloced buffer. */

   free ( ptr ) ;
   break ;

default :

   /* File appended, just close it. */
```

```
         fclose ( fp ) ;
         sm_ferr_reset ( "File appended" ) ;
         break ;
      }
      return ( 0 ) ;
   }
```

Automatic Screen

The following screen function, intended as the application's automatic screen function, maintains information on how long screens are open, and the total amount of time they are active. Note the use of the `P_USER` pointer, a general purpose pointer that you can manipulate, which is associated with an open screen.

This function keeps track of the length of time that the user has spent with a screen open and active. It is intended to be installed as the default screen function for an application. Note that in the example, the times are shown on the status line, but they could be logged to a file for time management analysis.

For this function to operate correctly, the setup variable `EXPHIDE_OPTION` must be set to `ON_EXPHIDE`, so Panther calls functions on screen overlay and reexposure.

The `time()` call used in this function is ANSI C. On UNIX platforms it returns the number of seconds elapsed since January 1, 1970, GMT.

```
/* Include Files */
#include "smdefs.h"    /* screen manager Header File */
#include "smglobs.h"   /* screen manager Globals */
#include <time.h>      /* ANSI time() Header File */


/* Data structure to hold aggregate times by screen */

struct my_info
{
   time_t opentime ; /* Time screen was opened */
   time_t acttime ;  /* Time screen was activated */
   double usedtime;  /* Aggregate time active */
   double totaltime ;/* Aggregate time open */
};

int
auto_sfunc ( name, context )
char *name ;  /* Screen Name */
int context ; /* Context for function call */
```

```
{
   struct my_info *my_info_ptr ;    /* Time buf pointer */
   char *action_verb =              /* Text of context */
   "inspecting" ;
   time_t current_time ;
   int do_free = 0 ;               /* Flag, set to free
                                       memory */

   char msg_buf[ 128 ] ;           /* Message buffer */

   /*
   * We make assumptions here: screens that are not named
   * are unimportant and should not have logging done.
   * This will exclude dynamically created message
   * windows.
   */

   if ( ( ! name ) || ( ! *name ) )
   {
      return ( 0 ) ;
   }

   /* Get the current time. ( ANSI Standard call ) */
   current_time = time ( (time_t *)0 ) ;

   /* Get the pointer to time structure
      associated with this screen */
   my_info_ptr = (struct my_info *)sm_pinquire ( P_USER ) ;

    /* Figure out which context we are called in. */
   if ( context & K_ENTRY )
   {
      if ( context & K_EXPOSE )
      {
         /*
         * Screen exposed (activated) when
         * overlying window was closed.
         * Set context string verb and
         * add to the aggregate open time.
         */

         action_verb = "activating" ;
         my_info_ptr->totaltime =
             my_info_ptr->totaltime +
             difftime ( current_time,
                 my_info_ptr->opentime ) ;


      }
      else
      {
         /* Screen opened. */
         action_verb = "opening" ;

         /* Allocate memory for time structure */
         my_info_ptr =
             (struct my_info *)
             malloc ( sizeof (
             struct my_info ) ) ;
         if ( ! my_info_ptr )
         {
            sm_ferr_reset ( "No memory" ) ;

            sm_cancel ( 0 ) ;
         }

         /* Associate the buffer with screen */
         sm_pset ( P_USER, (char *)my_info_ptr ) ;

         /* Set initial time values */
         my_info_ptr->opentime = current_time ;
         my_info_ptr->usedtime = 0 ;
         my_info_ptr->totaltime = 0 ;
      }

      /* Set initial value of aggregate active time */
      my_info_ptr->acttime = current_time ;
   }
   else
   {
      if ( context & K_EXPOSE )
      {
         /* Screen overlaid with window. */
         action_verb = "deactivating" ;
      }

      else
      {
         /* Screen closed. */
         action_verb = "closing" ;
          /* Set flag to free the time structure */

         do_free = 1 ;

      }

      /* Calculate new aggregates. */
```

```
            my_info_ptr->usedtime =
                my_info_ptr->usedtime +
                difftime ( current_time,
                my_info_ptr->acttime ) ;

            my_info_ptr->totaltime =
                my_info_ptr->totaltime +
                difftime ( current_time,
                my_info_ptr->opentime ) ;

         }

         /* Format the message. */

         sprintf ( msg_buf, "Now %s screen %s."
               "  Seconds active: %.1f."
                   "  Seconds open: %.1f.",
             action_verb, name,
             my_info_ptr->usedtime,
             my_info_ptr->totaltime ) ;

         /* If time structure memory should be freed, free it. */

         if ( do_free )
         {
            free ( my_info_ptr ) ;
         }

         /* Output the message. Could be to log file,
            here it is to stat line */

         sm_ferr_reset ( msg_buf ) ;

         return ( 0 ) ;
      }
```

Automatic Widget

This section has two sample functions:

- `auto_ffunc` puts general information about the current widget on the status line.
- `memoval` uses a widget's memo edits to pass non-standard information to that widget's automatic widget function.

Example 1

This function puts general information about the current widget on the status line. This function is installed as the automatic widget function in a Panther application; it is called on entry, exit, and validation for all widgets.

On widget entry, the function places information about the widget on the status line: its name, if any, number, and occurrence offset. If the widget is a selected member of a group—for example, radio buttons or a list box—the status line shows the text of the selected widget, the group name, and the group occurrence.

```
/* Include Files */
#include "smdefs.h"      /* screen manager Header File */

int
auto_ffunc ( f_number, f_data, f_occurrence, context )
int f_number ;            /* Field Number */
char *f_data ;            /* Field Data */
int f_occurrence ;       /* Array Index */
int context ;               /* Context Bits */

{
   char *f_name ;        /* Field Name */
   char *g_name ;        /* Group Name */
   char *slct ;            /* selected or deselected */
   int g_occurrence ;    /* Group Number */
   char stat_line[ 128 ];/* Status line string */

   /* If called on field exit, clear the status line. */
   if ( context & K_EXIT )
   {
      sm_setbkstat ( "", WHITE ) ;
   }

   /* If called on entry, format and display status line */
   else if ( context & K_ENTRY )
   {

      /* Obtain the field name */
      f_name = sm_name ( f_number ) ;

      /* Format the status line */
      if ( f_name && *f_name)
         sprintf ( stat_line, "Current Field: "
             "%s[%i] ( #%i[%i] )",
             f_name, f_occurrence,
             f_number, f_occurrence ) ;
      else
         sprintf ( stat_line,
            "Current Field: #%i[%i]",
            f_number, f_occurrence ) ;

      /* Display the status line */
      sm_setbkstat ( stat_line, BLUE | HILIGHT ) ;
   }

   /*
   * If we get here, it is neither entry nor exit so it must
   * be validation. In this case, see if the field is the
   * member of a group. If it is, the validation function
   * was called because the field was selected, or in the
   * case of checklists, deselected. Note that
   * menu selection events will not be flagged, because
   * menus are not groups.
   */

   else if ( g_name = sm_o_ftog ( f_number,
         f_occurrence,
         &g_occurrence ) )

   {
      /* Determine if selected or deselected */

      if ( sm_isselected ( g_name, g_occurrence ) )
         slct = "selected" ;
      else
         slct = "deselected" ;

      /* Format and print status line message */

      sprintf ( stat_line, "%s %s, group %s[%d]",
          f_data, slct, g_name, g_occurrence ) ;

      sm_setbkstat ( stat_line, BLUE | HILIGHT ) ;
   }

   /* Return code of zero means that everything is fine. */

   return ( 0 ) ;
}
```

Example 2

This second example of an automatic widget function shows how to use a widget's memo edits to pass non-standard information to that function. This function validates each widget against the contents of its memo edits.

This function is to be installed as a non-prototyped widget validation function in a Panther application, either on the

FIELD_FUNC list or as the

DFLT_FIELD_FUNC.

The function validates widgets according to a list of values that are found in the first memo text edit. Possible values in the memo text edit are separated by spaces.

```
/* Include Files */

#include "smdefs.h"       /* screen manager Header File */

int
memoval ( f_number, f_data, f_occurrence, context )
int f_number ;             /* Field Number */
char *f_data ;             /* Field Data */
int f_occurrence ;         /* Array Index */
int context ;              /* Context Bits */

{
   char *memo_text ;      /* Memo text string */
   char *token_ptr ;      /* Token */
   char msg[ 128 ] ;      /* message string */

   /* If called on field entry or exit, or if already
      validated, or if empty, just exit right off. */

   if ( ( context & K_EXIT  ) ||
        ( context & K_ENTRY ) ||
        ( context & VALIDED ) ||
        ( ! *f_data ) )

   {
      return ( 0 ) ;
   }

   /* Get the first memo text edit string. */

   if ( ! ( memo_text = sm_edit_ptr ( f_number, MEMO1 ) ) )
   {
      /* There is no memo text edit string. */
      return ( 0 ) ;
   }

   /* Duplicate the string. (Note: pass over the two length
      bytes returned by sm_edit_ptr) */
   if ( ! ( memo_text = strdup ( memo_text + 2 ) ) )
   {
      /* Memory allocation error. */
      return ( 0 ) ;
   }

   /* Cycle down the memo text string grabbing tokens.
      If we have a match, break out of loop. */

   for ( token_ptr = strtok ( memo_text, " " ) ;
       token_ptr && strcmp ( token_ptr, f_data ) ;
       token_ptr = strtok ( NULL, " " ) ) ;

   /* Free up memory. */
   free ( memo_text ) ;

   /* If we found matching token, validate OK. */
   if ( token_ptr )
      return ( 0 ) ;

   /* Error condition. Create error string. */

   sprintf ( msg, "Invalid value %s in field. "
       "Valid values are: %s.", f_data,
       sm_edit_ptr ( f_number, MEMO1 ) + 2 ) ;

   sm_ferr_reset ( 0, msg ) ;

   /* Return and reset cursor. */

   return ( 2 ) ;
}
```

Demand Widget

The following local widget functions can be called by individual widgets to perform initialization and validation based on external criteria:

Two widget functions to include on the widget function list are defined here. The first one, `fentry`, initializes the value in a widget provided that it has not changed since the screen was opened. The second one, `fvalid`, validates the contents of a widget. The functions that retrieve the initialization data and lookup the validation data are externally defined and are application-specific.

```
/* Include Files */

#include "smdefs.h"   /* screen manager Header File */
```

```
/* Externally defined functions */
extern char *do_my_initialize ( ) ; /* Get data for field
                                          initialization */
extern int my_lookup ( ) ;          /* Lookup data for field
                                          validation */

int
fentry ( f_number, f_data, f_occurrence, f_context )
int f_number ;       /* Field Number */
char *f_data ;       /* Field Data */
int f_occurrence ;   /* Array Index */
int f_context ;      /* Context bits */

{
   /* Initialize if the field has not been modified
      since the screen was opened. */

   if ( ! ( f_context & MDT ) )
   {
      sm_putfield ( f_number, do_my_initialize ( ) ) ;
   }

   return ( 0 ) ;
}

int
fvalid ( f_number, f_data, f_occurrence, f_context )
int f_number ;        /* Field Number */
char *f_data ;        /* Field Contents */
int f_occurrence ;    /* Occurrence number for field */
int f_context ;       /* Context bitmask */

{
   char msg_buf[ 80 ];/* Message line buffer */

   /* If the field is already valid, merely return. */
   if ( f_context & VALIDED )
      return ( 0 ) ;

   /* If the field is invalid based on external
      lookup, return error. */
   if ( my_lookup ( f_data ) )
   {
      /* Error, so reposition field. */

      sm_gofield ( f_number ) ;
```

```
            sprintf ( msg_buf, "Invalid data %s.", f_data ) ;
            sm_ferr_reset ( 0, msg_buf ) ;

            /* Return code of 1 indicates validation fail */

            return ( 1 ) ;
         }
          return ( 0 ) ;
      }
```

Automatic Group

The group function `auto_gfunc` is installed as the automatic group function—that is, a function that is called whenever group entry, exit, or validation occurs. On entry, this function installs the keychange function `keychg`, which lets users select group widgets by pressing the X key. On group exit, `auto_gfunc` uninstalls `keychg`.

Note that preexisting keychange functions should be stacked by `auto_gfunc`. `keychg` also chains existing keychange functions along, but it is assumed that they are written in C. Preexisting keychange functions in some other supported 3GL language may not be properly chained by this function.

For a more extended example of keychange functions, see page 44-79.

```
/* Include Files */
#include "smdefs.h"   /* screen manager Header File */
#include "smkeys.h"   /* screen manager Logical Keys */

static int keychg ( ) ;
static struct fnc_data o_keychg ;   /* Old keychg */
static struct fnc_data *fnc_ptr ;   /* Event Pointer */
static struct fnc_data keychg_struct/* New keychg */
      = { 0, keychg, 0, 0, 0, 0 };

int
auto_gfunc ( name, context )
char *gp_name ;           /* Group Name */
int context ;             /* Context bits */

{
   /* If called on group entry.... */
   if ( context & K_ENTRY )
   {
      /* Install the new keychange function */
      fnc_ptr = sm_install ( KEYCHG_FUNC,
          &keychg_struct,
          (int *)0 ) ;

      /* If there was an old one, store it away. */
      if ( fnc_ptr )
      {
         memcpy ( (char *)&o_keychg,
             (char *) fnc_ptr,
             sizeof ( struct fnc_data ) ) ;
      }

      else
      {
         memset ( (char *)&o_keychg, 0,
             sizeof ( struct fnc_data ) ) ;
      }
    }

   /* If called on group exit...... */
   else if ( context & K_EXIT )
   {
      /* If there was an old keychange function */
      if ( fnc_ptr )
      {
         /* Re-install it. */
         sm_install ( KEYCHG_FUNC, &o_keychg,
             (int *)0 ) ;
      }

      else
      {
         /* Get rid of the current one anyway. */
         sm_install ( KEYCHG_FUNC, NULL,
             (int *) 0 ) ;
      }
   }
   return ( 0 ) ;
}

static int
keychg ( key )
int key ;

{
   /* If there was an old keychange function ..... */
   if ( o_keychg.fnc_addr )
   {
      /* Chain the old keychange function. */
      key = ( o_keychg.fnc_addr )( key ) ;
      /* WARNING: This is not completely general, since
         old keychange functions not written in C
         may not be called properly. */

   }

   /*
   * Now do the new keychange. Basically, we want to select
   * group members by typing "x", move the cursor to the
   * next group member immediately after selection, and have
   * the NL key move to the next selection.
   */

   switch ( key )
   {
   case 'x' :
   case 'X' :
      key = NL ;
      break ;

   case NL :
      key = ' ' ;
      break ;
   }

   return ( key ) ;

}
```

External Help

`sm_PiXmDynaHook` is a help function that Panther uses to invoke its own help facilities. Use the External Help Tag property of screens, menus, and screen-resident widgets to specify help context identifiers. This identifier is passed to the help function when the user invokes help from an application component.

The following help driver is supplied with Panther; it invokes context-sensitive help from the screen editor.

```
/*** sample client code for dyna help server ***/
/** Includes **/
#include "smmach.h"
#include "smproto.h"
#include "smxmuser.h"
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
```

```
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <sys/wait.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <X11/StringDefs.h>
#include "xmhelphk.h"

/* typedef */
typedef struct PiXmDynaPath_s PiXmDynaPath_t;
struct PiXmDynaPath_s
{
   char *pszDynaCollection;
   char *pszDynaBook;
};

/** statics **/
static Atom xaServer  = (Atom)0;
static Atom xaRequest = (Atom)0;
static PiXmDynaPath_t dynaPath = {NULL, NULL};
static XtResource xresDyna[] =

{
   { "helpPath", "HelpPath", XtRString, sizeof(char *),
     XtOffsetOf(PiXmDynaPath_t, pszDynaCollection),
      XtRString,
     "/u/apps/ebt22" },
   { "editorHelpFile", "EditorHelpFile", XtRString,
      sizeof(char *), XtOffsetOf
      (PiXmDynaPath_t, pszDynaBook), XtRString, "editors" },
};

int sm_PiXmDynaHook PROTO((char *));
static int PiXmInitHelp PROTO((Display *));
static void PiXmSendMsg PROTO((Display *, Window, char *));
/*

NAME
   The event function to the Dyna Server. Takes  "Tag" string
   and makes a help server request. However if the "Tag" is
   the string "SM_RESERVED_QUIT_TAG" then server is killed.

SYNOPSIS
   iRetVal = sm_PiXmDynaHook(pszTag);
   char *pszTag;        The "Tag" identification string
   int iRetVal;

DESCRIPTION
   0) If  tag is "SM_RESERVED_QUIT_TAG" the quit. Otherwise.
```

```
    1) Get the path of the help file
    2) Initialize the server (if needed)
    3) Build the server request
    4) send the request

RETURNS
    Returns
        PI_ERR_NONE on success
        PI_ERR_NO_MORE on failure
*/

int
sm_PiXmDynaHook PARMS((pszTag))
LASTPARM(char *pszTag)

{
    char pszMessage[512];
    Display *dpy = sm_xm_get_display();
    Boolean bQuiting = strcmp(pszTag, "SM_RESERVED_QUIT_TAG");

    if (dynaPath.pszDynaBook == NULL)
    {
        XtGetApplicationResources(sm_xm_get_base_window(),
            &dynaPath, xresDyna, XtNumber(xresDyna), NULL, 0);
    }

        if (!PiXmInitHelp (dpy))
        return(PI_ERR_NO_MORE);
        sprintf(pszMessage,
        "command=ebt-link collection=%s book=%s
        target=ancestor(ancestor(idmatch('Tagname','%s')))
        stylesheet=fulltext.v showtoc=true",
        dynaPath.pszDynaCollection, dynaPath.pszDynaBook,
        pszTag);
        PiXmSendMsg (dpy, DefaultRootWindow(dpy), pszMessage);
    return(PI_ERR_NONE);
}

/*

NAME
    PiXmInitHelp - Start the server if needed. Set server atoms.

SYNOPSIS
    iRetVal = PiXmInitHelp(dpy);
    Display *dpy;
    int iRetVal;

DESCRIPTION
     Start the server if needed. Set the server atoms.
```

```
RETURNS
    Returns
        PI_ERR_NONE on success
        PI_ERR_NO_MORE on failure
*/

static
int
PiXmInitHelp PARMS((display))
LASTPARM(Display *display)

{
    int iPid;
    int iDummy;
    char *pszShellPath;      /* pointer to SHELL environment var */
    char *pszShell;          /* the last segment of the path     */
    void (*pfuncIntr)(), (*pfuncQuit)(), (*pfuncTstp)();
    char *server_name = "SM_JAM_DYNA_HELP_SERVER";
    char *selection_name = "SM_JAM_DYNA_HELP_SELECTION";
    char *request_name = "SM_JAM_DYNA_HELP_REQUEST";
    int iRetVal = 1;
    static Boolean bCreatedServer = FALSE;
        XInternAtom (display, selection_name, False);
    xaServer = XInternAtom (display, server_name, False);
        if (!bCreatedServer)
        XSetSelectionOwner(display, xaServer, None,
        CurrentTime);
    if ((!bCreatedServer) || XGetSelectionOwner(display,
            xaServer) == None)
    {
#ifdef SIGTSTP
#define SIGNAL(a,b) signal(a,b)
            pfuncTstp = SIGNAL(SIGTSTP, SIG_DFL);

#else

#define SIGNAL(a,b)

#endif

            /* see if there is a SHELL variable */
        pszShellPath = getenv ("SHELL");
            if (!pszShellPath || !pszShellPath[0])
            pszShellPath = "/bin/sh";
            if (!(iPid = fork()))
        {
            pszShell = strrchr(pszShellPath, '/');
            if (!pszShell || !pszShell[1])
                pszShell = pszShellPath;
                execlp (pszShellPath, pszShell, "-c",
                "xmjxhelp", (char *)0);
                exit (-1);
        }
    }

        bCreatedServer = TRUE;
        pfuncIntr = signal(SIGINT, SIG_IGN);
    pfuncQuit = signal(SIGQUIT, SIG_IGN);
        while (XGetSelectionOwner(display, xaServer) == None)
    {
        if(waitpid(iPid, &iDummy, WNOHANG))
        {
            /* server failed */
            iRetVal = PI_ERR_NO_MORE;
            break;
        }

        else
        {
            sleep(1);
        }
    }

        signal (SIGINT, pfuncIntr);
    signal (SIGQUIT, pfuncQuit);
    SIGNAL (SIGTSTP, pfuncTstp);
        xaRequest = XInternAtom(display, request_name, False);
    return(iRetVal);

}

/*

NAME
    PiXmSendMsg - Send the msg request to the help server.

SYNOPSIS
    PiXmSendMsg(dpy, xwin, pszMsg);
    Display *dpy;
    Window xwin;
    char *pszMsg;

DESCRIPTION
    Send the msg request to the help server.
*/

static
void
PiXmSendMsg PARMS((dpy, xwin, pszMsg))
PARM(Display *dpy)
```

```
PARM(Window xwin)
LASTPARM(char *pszMsg)

{
       XChangeProperty(dpy, xwin, xaRequest, XA_STRING, 8,
       PropModeReplace, (unsigned char *) pszMsg,
           strlen(pszMsg)+1);
    XConvertSelection(dpy, xaServer, XA_STRING, xaRequest,
           xwin, CurrentTime);
    XFlush(dpy);
}
```

Timeout

`screen_saver` is a timeout function that acts as a screen saver that is invoked after ten minutes of keyboard inactivity. The same function restores the screen when a key is typed.

```
/* Include files */
#include "smdefs.h"

int screen_saver( int why_called )
{
   if ( why_called == TF_TIMEOUT ) /*clear the screen
                                     after timeout */

   {
      sm_clrviscreen ( );
   }
   else if ( why_called == TF_RESTART ) /*restore screen
                                          after key hit */

   {
      sm_rescreen ( );
   }

   /* Returning STOP_CALLING means this function is not
    * called again every ten minutes
    */

   return ( TF_STOP_CALLING )
}
```

Key Change

The following key change function intercepts each occurrence of the user entering an exclamation point or striking the EXIT key.

This application keychange function causes sm_getkey to intercept two keys, the exclamation point and the logical EXIT key. When the user types an exclamation point, this function asks if an operating system shell is wanted. If so, a shell is provided. If the user types EXIT, the function ensures that the user really wants to EXIT before returning the EXIT back to sm_getkey.

Note that if the user escapes to the shell or does not want to EXIT, the keychange function swallows the keystroke. If the user does not want the shell or wants to EXIT, the keystroke is passed back to sm_getkey.

Note also preprocessor directives on whether or not the Panther executive is in use. If the executive is in use, we do not query about the EXIT if there are control strings associated with EXIT. Also, we can use the standard Panther operating system escape.

```
/* Include Files */
#include "smdefs.h"  /* screen manager Header File */
#include "smkeys.h"  /* screen manager Logical Keys */

#define EXIT_CONFIRM "Do you want to EXIT? (y/n)"
#define SHELL_CONFIRM "Do you want to go to OS? (y/n)"

int keychg ( int the_key )/* Key read from keyboard by
                           * sm_getkey */

{
   static int recursive ;  /* Flag ensuring no recursion. */

   /* First ensure that we are not called recursively */
   if ( recursive ) return ( the_key ) ;

   /* Set recursive flag */
   recursive++ ;

   /* Based on the key read from the keyboard..... */
   switch ( the_key )

   {
   case EXIT:
      /*
      * If the read key is an EXIT, make sure that there are
      * no control strings associated with EXIT and confirm
      * that the user really wants to EXIT. If the user does
```

```
          * not want to, set the key to zero. The PROL_EXECUTIVE
          * macro is not defined in any Panther header file. It
          * is used here to distinguish between applications that
          * use the Panther executive and those that don't.
          */
          if (
#ifdef PROL_EXECUTIVE
              ! sm_getjctrl ( EXIT, 0 ) &&
              ! sm_getjctrl ( EXIT, 1 ) &&
#endif
              ( sm_query_msg ( EXIT_CONFIRM )
              == 'n' ) )
          {
             the_key = 0 ;
          }
          break ;

       case '!':
          /*
          * If the read key is an exclamation point, confirm
          * that the user really wants to escape to the shell
          * If so, escape to the shell and gobble up the key.
          * If not, merely pass the key on back
          */
          if ( sm_query_msg ( SHELL_CONFIRM ) == 'y' )
          {
             sm_leave ( ) ;

             /* SHELL UNDER UNIX */
             system ( "sh -i" ) ;

             sm_return ( ) ;
             sm_rescreen ( ) ;

             the_key = 0 ;
          }
          break ;
       }

       /* Clear the recursion flag. */
       recursive = 0 ;

       /* Pass the key back up. (If it is changed to zero,
          we gobbled it.) */
       return ( the_key ) ;
}
```

Error

The following error function writes all user messages to a log file.

```
#include <smdefs.h>

/* log all messages sent to the user to the file "err.txt" */

int myerr(int msgno, char *msgtxt, int quiet_mode)
{
    FILE *fp;

    /* by default, use 'msgtxt' param & no prepended
     * "ERROR " string
     */
    char *err_msg = msgtxt;
    char *quiet_txt = "";

    /* if msgno != -1, retrieve msg text from msg file */
    if (msgno != -1)
        err_msg = sm_msg_get(msgno);

    /* if called via the 'quiet' variety of error function */
    if (quiet_mode)
        quiet_txt = "ERROR: ";

    fp = fopen("err.txt", "a+");

    if (fp == NULL) {
        perror("error opening 'err.txt'");
        exit(1);
    }

    fprintf(fp, "myerr: %s'%s'", quiet_txt, err_msg);

    fclose(fp);
    return 0;
}
```

Insert Toggle

The following example shows a function that displays the current insert/overwrite mode at the end of the status line. This function is installed as the `INSCRSR_FUNC` function, called whenever Panther moves from insert to overstrike mode or vice versa. The status line displays `INS` when in insert mode, and `OVR` when in overstrike mode.

This routine assumes that cursor position display is not in use. You may also need a STAT_FUNC function for this, as Panther overwrites the status line with messages, thus destroying the INS/OVR message.

The last column of the status line is not written; Panther does not permit writing to the last position of a screen if it causes automatic hardware scrolling.

```
/* Include Files */

#include "smdefs.h"  /* screen manager Header File */

/* Buffer Sizes */

#define STAT_LINE_LEN 80

int inscrsr ( int enter_ins_mode );
                 /* enter_ins_mode is non-zero if about to
                  * enter insert mode, zero if about
                  * to enter overstrike mode
                  */
{
   if ( enter_ins_mode )
      sm_d_msg_line ( "INS", 0) ;
   else
      sm_d_msg_line ( "OVR", 0) ;
   return ( 0 ) ;
}
```

Initialization and Reset

The following code shows an example of initialization and reset functions. Note that most of the initialization need not be done in the initialization event. It could be done before `sm_initcrt` is called.

The two functions below, `uinit` and `ureset`, are to be installed as the initialization and reset functions respectively.

- `uinit` is used to initialize the automatic variable `start_time`. Then `uinit` asks the user to enter a terminal type, and passes the string back to `sm_initcrt` for processing. Finally, `uinit` establishes error handling that causes the application to terminate gracefully on a number of software signals.

- `ureset` calculates the elapsed time that the user has been in the application and prints it to the terminal.

Note that `ssignal` is ANSI C. The signals `SIGINT`, `SIGABRT`, and `SIGTERM` are all part of ANSI C and the Posix standard, and are meaningful on most but not all platforms.

```
/* Include Files */

#include "smdefs.h"   /* screen manager Header File */
#include <signal.h>   /* software signals */

 static time_t start_time ; /* Application start time */

int
uinit ( term )
char * term ;    /* 30-byte buffer with terminal type */
{
   char * ptr ;

   /* Determine current time as starting time. */
   start_time = time ( (time_t*)0 ) ;

   /* Get terminal type from user. (If nothing entered,
      system will use the environment.) */
   printf ( "Please enter terminal type: " ) ;

   if ( ! fgets ( term , 29 , stdin ) ) * term = '' ;
   term[ 29 ] = '' ;

   if ( ptr = strchr ( term , '' ) ) * ptr = '' ;

   /* Establish necessary signal handling. */
   ssignal ( SIGINT , sm_cancel ) ;
   ssignal ( SIGABRT , sm_cancel ) ;
   ssignal ( SIGTERM , sm_cancel ) ;
   return ( 0 ) ;
}

int
ureset ( )
{
   int hours , minutes , seconds ;

   /* Determine elapsed time since start of application
      and calculate hours, minutes, and seconds
      elapsed. */

   seconds = (int)difftime ( time ( (time_t *)0 ),
         start_time ) ;
   minutes = seconds / 60 ;
   seconds %= 60 ;
   hours = minutes / 60 ;
   minutes %= 60 ;
   /* Print out time report. */

   printf ( "Application active for %d hours, %d minutes, "
       "%d seconds.", hours, minutes, seconds ) ;

   return ( 0 ) ;
}
```

Record and Playback

The following example shows how record and playback might work together in a regression test.

The two functions record and play implement a simple mechanism for recording and later playing back keystrokes in a Panther application. The keystrokes are recorded to and played back from a file. The interval in seconds between keystrokes is also saved so that the playback function can pause to simulate real user behavior.

The following lines can be included in the main function to allow for conditional record and playback, assuming that the first parameter passed to the program was an optional indicator for record or playback:

```
if ( argc > 1 )
{
     switch ( argv[ 1 ][ 0 ] )
    {
       case 'r' :
       case 'R' :
          sm_install( RECORD_FUNC, &record_struct,(int *)0);
             break ;
       case 'p' :
       case 'P' :
          sm_install ( PLAY_FUNC, &play_struct, (int *)0 ) ;
             break ;
    }
}
```

It is preferable for the main function to initialize the variable `r_time` rather than counting on this record/playback system to do it. Used as written, the interval before the very first key that the user types is not accurately recorded, and hence not accurately played back.

```
/* Include Files */
```

```
#include "smdefs.h"    /* screen manager Header Files */
static int intbuf[2] ; /* Buffer for read/write of
                        * keystroke data */

static FILE *fp ;        /*File pointer for keystroke file */

static time_t r_time ;   /*Time first character was gotten */

static time_t c_time ;   /* Current time;
                          * interval=difftime(c_time, r_time)
                          */

static char key_file[ ]  /* Name of keystroke file */
= "recplay.key" ;

int
record ( key )
int key ;                /* Key to be recorded */
{
   /* If the file has not been opened, open it and
      initialize r_time */
   if ( ! fp )
   {
      /* Set the initial time. */
      r_time = time ( (time_t *)0 ) ;

      /* Open file */
      fp = fopen ( key_file, "w" ) ;

      /* Turn on record/playback system */
      sm_keyfilter ( 1 ) ;
   }

   /* Get the current time */
   c_time = time ( (time_t *)0 ) ;

   /* Store the key to record in the data buffer */
   intbuf[ 0 ] = key ;

   /* Store the time interval in the data buffer */
   intbuf[ 1 ] = floor ( difftime ( c_time, r_time )
       + 0.5 ) ;

   /* Now write the data buffer to the keystroke file */
   if ( ( ! fp ) ||
       ( fwrite ( (char *) intbuf, sizeof ( int ),
       2, fp ) != 2 ) )
   {
      /* Write failed. Close everything down.... */
      fclose ( fp ) ;
      fp = NULL ;
      intbuf[ 0 ] = 0 ;
      sm_keyfilter ( 0 ) ;
      sm_ferr_reset ( "Recording Terminated..." ) ;
   }

   return ( 0 ) ;
}


int
play ( )
{
   /* If the file has not been opened, open it and
      initialize r_time */
   if ( ! fp )
   {
      r_time = time ( (time_t *)0 ) ;
      fp = fopen ( key_file , "r" ) ;
      sm_keyfilter ( 1 ) ;
   }

   /* Now read the keystroke file, one keystroke into
      the data buffer */
   if ( ( ! fp ) ||
       ( fread ( (char *) intbuf, sizeof ( int ),
       2, fp ) != 2 ) )
   {
      /* Read failed. Close everything down.... */
      fclose ( fp ) ;
      fp = NULL ;
      intbuf[ 0 ] = 0 ;
      sm_keyfilter ( 0 ) ;
      sm_ferr_reset ( "Playback Terminated...." ) ;
      return ( 0 ) ;
   }

   /* Get the current time */
   c_time = time ( (time_t *)0 ) ;

   /* Decrement interval from data buffer by measured
      interval */
   intbuf[ 1 ] -= floor ( difftime ( c_time, r_time )
       + 0.5 ) ;

   /* Sleep some more if we should. */
   if ( intbuf[ 1 ] > 0 )
   {
      sm_flush ( ) ;
      sleep ( intbuf[ 1 ] ) ;
   }

   /* Return the key to sm_getkey for processing */
   return ( intbuf[ 0 ] ) ;
}
```

Control

The following example shows two closely related functions that you can include on a control function list. The `mark_low` function marks all widgets on the current screen with numeric values less than zero with an attribute change. The function `mark_high` marks all widgets on the current screen with numeric values higher than 1000. The same functionality is duplicated as the prototyped function `mark_flds` (see page 44-52).

Note that both `mark_low` and `mark_high` call the static function `mark_flds` which actually does the work. This may seem like unnecessary indirection, but it means that the control strings used are very simple, as shown here:

```
^mark_low
^mark_high
/* Include Files */
#include "smdefs.h"   /* screen manager Header File */
#include "smglobs.h"  /* screen manager Globals */

/* Macro Definitions... */
/* Attributes used to mark fields */

#define MARK_ATTR REVERSE | HILIGHT | BLINK
#define MARK_GT 1    /* Indicates "Greater Than" */
#define MARK_LT -1   /* Indicates "Less Than " */

static int mark_flds ( ) ;


int
mark_low ( ctrl_str )
char *ctrl_str;/* control string text passed by Panther */
{
   /* Mark all fields less than zero */
   return ( mark_flds ( 0, MARK_LT ) ) ;
}

int
mark_high ( ctrl_str )
```

```
char *ctrl_str;/* control string text passed by Panther*/
{
   /* Mark all fields greater than one thousand */
   return ( mark_flds ( 1000, MARK_GT ) ) ;
}

static int
mark_flds ( bound, operator )
int bound ;          /* Boundary on fields to mark */
int operator ;       /* Operator, MARK_GT or MARK_LT */
{
   int fld_num ;     /* Field Number */
   int num_of_flds ; /* Number of Fields */

   /* Determine number of fields */
   num_of_flds = sm_inquire ( SC_NFLDS ) ;

   /* Cycle through all the fields on the screen */
   for ( fld_num = 1 ; fld_num <= num_of_flds ; fld_num++ )
   {
      /* Depending on the operator... */
      switch ( operator )
      {
      case MARK_GT:
         /* Mark fields that are
            greater than the
            given bound. */
         if ( sm_dblval ( fld_num )
             > ( double ) bound )
         {
            sm_chg_attr ( fld_num,
                MARK_ATTR ) ;
         }

         break;

      case MARK_LT:
         /* Mark fields that are less
            than the given bound */
         if ( sm_dblval ( fld_num )
             < ( double ) bound )
         {
            sm_chg_attr ( fld_num,
                MARK_ATTR ) ;
         }
         break;
      }
   }
   return ( 0 ) ;
}
```

The next example shows how a number of entries in a control function list might map to the same function, `report`, which uses the identifying string as an implied first argument. Significant argument processing is done in this example.

- `report` creates a report about the state of the current widget, screen, window stack, or display. The report can be appended to a file, passed as an argument to an operating system command, piped to an operating system command, or displayed in a message window.
- `report` is installed under four names in the `CONTROL_FUNC` function list for Panther control functions. When a control string calling this function is invoked, the entire control string is passed as an argument to this function. The name used to invoke the function is an implied argument and specifies which report to generate: widget, screen, window stack, or display. The remainder of the control string specifies what to do with the report output. This can be one of the following four categories:

1. If there is nothing on the control string following the name, the report is printed in a popup message window. For example, the following control string will generate a report about the current widget and display the report in a popup message window:

   ```
   ^rep_field
   ```

2. If the arguments start with an exclamation point (!), the rest of the control string is taken to be an operating system command. In this case, a temporary file with the report will be created, and the file name will be appended to the operating system command. However, if the operating system command has a tilde (~) in it, the tilde will be replaced with the name of the file before the command is invoked. In any event the file is deleted after the command is invoked. Two example control strings that would cause a screen report to be printed on a UNIX system are shown below:

   ```
   ^rep_screen !lp -c -s
   ```

   ```
   ^rep_screen !lp -c ~ > /dev/null 2>&1
   ```

3. If the arguments start with a vertical bar (|), the rest of the control string is taken to be an operating system command. In this case, however, the report will be created as the standard input of the specified command. Many operating systems

call this piping. The example shown here will cause a window stack report to piped through the UNIX command tail and printed, so that only 20 lines of output will be printed:

```
^rep_wstack |tail -20 | lp -c -s
```

4. If the arguments do not start with a vertical bar or with an exclamation point, the assumption is that it is a file that is named. The file will be created if it does not exist, or appended to if it does exist. The following example will append a display terminal report to the file `report.fil`:

```
^rep_term report.fil
```

This function installation is preceded with the following definitions and declarations, commonly found in `funclist.c`:

```
extern int report ( ) ;
struct fnc_data report_funcs[] = {
   SM_OLDFNC( "rep_field", report ),
   SM_OLDFNC( "rep_screen", report ),
   SM_OLDFNC( "rep_wstack", report ),
   SM_OLDFNC( "rep_term", report )
} ;

int report_count = sizeof ( report_funcs )
                            / sizeof ( struct fnc_data ) ;
```

The actual installation of the function is done with the following call to `sm_install`, usually found in `sm_do_uinstalls`, defined in `funclist.c`:

```
sm_install ( CONTROL_FUNC, report_funcs, &report_count ) ;
```

Note that the function list has four function entries with different names, all of which refer to the same function pointer. In the case of `CONTROL_FUNC` functions, the entire control string is passed to the called function in a string, the name used to invoke the function can—and in this case does—serve as an implied argument.

```
/* Include Files */

#include "smdefs.h"    /* screen manager Header File */
#include "smglobs.h"   /* screen manager Globals */

int
report ( report_type )
char *report_type ;   /* Text of invoking control
            string -- later truncated to the
             name of the desired report */
```

```
{
   char *report_out;   /* Report output designation */
   char *fn = NULL;    /* Name of output file */
   char *ptr, *ptr1;   /* Character pointers */
   char msg_buf[ 128 ];/* Message buffer */
   FILE *fp ;             /* File pointer for output */
   int size ;             /* Size of output file */
   int cur_no ;          /* Current field number */
   int select ;          /* Current window stack index */

   /* Set report output designator to control string
    *   arguments
    */
   for ( report_out = report_type ;
       *report_out && ( ! isspace ( UNSIGN(*report_out) ) ) ;
       report_out++ ) ;

   /* If control string has arguments.... */
   if ( *report_out )
   {
      /* Truncate the report type with a terminator */
      *report_out = '';

      /* Gobble up unnecessary white space */
      for ( report_out++ ;
          *report_out &&
          ( isspace ( *report_out ) ) ;
          report_out++ ) ;

      /* Based on what output type we designated: */
      switch ( *report_out )
      {
      case '!' :
         /* OS command. Open temp file */
         fn = tempnam ( NULL, "rprt" ) ;
         fp = fopen ( fn, "w" ) ;
         break ;

      case '|' :
         /* Pipe. Open the pipe */
         fp = popen ( report_out + 1,
             "w" ) ;
         break ;

      default :
         /* Other. Open the file */
         fp = fopen ( report_out, "a+" ) ;
         break ;
      }
      /* If we could not open the file, show error */
      if ( ! fp )
      {
         sprintf ( msg_buf,
             "Cannot open stream for %s.",
             report_out ) ;
         sm_ferr_reset ( msg_buf ) ;
         return ( -1 ) ;
      }
   }

   /* If no report output specified, open temp file for
      storing message window stuff. */
   else
   {
      fn = tempnam ( NULL, "rprt" ) ;
      fp = fopen ( fn, "w+" ) ;
      report_out = "" ;
   }

   /* Now, based on the report_type, which is the name
      with which the function was invoked, create
      the reports. Note that all newlines are
      preceded with spaces, this is so that in the
      case of the message windows we can replace
      all space-newlines with %N, the newline
      indicator for Panther windows. */
   if ( ! strcmp ( report_type, "rep_field" ) )

   {
      /* Output a field report */
      fprintf ( fp, "  Field Report: " ) ;

      /* Field Identifier and contents */
      fprintf ( fp, "FIELD: %d (%s[%d]) = %s ",
          cur_no = sm_getcurno ( ),
          sm_name ( cur_no ),
          sm_occur_no ( ),
          sm_fptr ( cur_no ) ) ;

      /* Field sizes */
      fprintf ( fp, "LENGTH: onscreen: %d "
          "Max: %d ",
          size = sm_finquire ( cur_no, FD_LENG ),
          sm_finquire ( cur_no, FD_SHLENG )
          + size ) ;

      fprintf ( fp, "# OCCURRENCES: onscreen: %d "
          "Max: %d ",
          sm_finquire ( cur_no, FD_ASIZE ),
          sm_max_occur ( cur_no ) ) ;
   }
   else if ( ! strcmp ( report_type, "rep_screen" ) )
   {
      /* Output screen report */
      fprintf ( fp, "  Screen Report: " ) ;

      /* Screen Name */
      fprintf ( fp, "SCREEN: %s ",
          sm_pinquire ( SP_NAME ) ) ;

      /* How much of screen is visible */
      fprintf ( fp, "%% VISIBLE IN VIEWPORT: %d ",
          100 *
          ( sm_inquire ( SC_VNLINE ) *
          sm_inquire ( SC_VNCOLM ) ) /
          ( sm_inquire ( SC_NCOLM ) *
          sm_inquire ( SC_NLINE ) ) ) ;
   }
   else if ( ! strcmp ( report_type, "rep_wstack" ) )
   {
      /* Output Window stack report */
      fprintf ( fp, "  Window Stack Report: " ) ;

      /* Cycle through all the windows. */
      for ( select = 0 ;
          sm_wselect ( select ) == select ;
          select++ )
      {
         /* Window number... */
         fprintf ( fp, " Window %d: ",
             select ) ;

         /* Screen name */
         fprintf ( fp, "screen: %s ",
             sm_pinquire ( SP_NAME ) ) ;

         /* Number of fields and groups */
         fprintf ( fp, "# of Fields: %d "
             "# of Groups: %d ",
             sm_inquire ( SC_NFLDS ),
             sm_inquire ( SC_NGRPS ) ) ;

         sm_wdeselect ( ) ;
      }
      sm_wdeselect ( ) ;
   }
   else if ( ! strcmp ( report_type, "rep_term" ) )
   {
      /* Output display terminal report */
      fprintf ( fp, "  Terminal Report: " ) ;

      /* Terminal Type */
      fprintf ( fp, "TERM TYPE: %s ",
          sm_pinquire ( P_TERM ) ) ;

      /* Display mode */
      if ( sm_inquire ( I_NODISP ) )
         fprintf ( fp, "DISPLAY OFF " ) ;
      else
         fprintf ( fp, "DISPLAY ON " ) ;

      /* Input mode */
      if ( sm_inquire ( I_INSMODE ) )
         fprintf ( fp, "INSERT MODE " ) ;
      else
         fprintf ( fp, "TYPEOVER MODE " ) ;

      /* Block mode */
      if ( sm_inquire ( I_BLKFLGS ) )
         fprintf ( fp, "BLOCK MODE " ) ;

      /* Physical display size */
      fprintf ( fp, "DISPLAY SIZE: %d x %d ",
          sm_inquire ( I_MXLINES ),
          sm_inquire ( I_MXCOLMS ) ) ;
   }

   else
   {
      /* Unrecognized report type */
      sprintf ( msg_buf, "Illegal report type %s",
          report_type ) ;
      sm_ferr_reset ( msg_buf ) ;
      fprintf ( fp, "%s  ", msg_buf ) ;
      return ( -3 ) ;
   }

   /* Once again, based on the type output... */
   switch ( *report_out )
   {
   case '|' :
      /* It was a pipe, so close it. */
      pclose ( fp ) ;
      sm_ferr_reset ( "Pipe successful" ) ;
      break ;
```

```
case '!' :
   /* It was an O/S command. Close file... */
   fclose ( fp ) ;

   /* Gobble up the exclamation point */
   report_out++;

   /* Look for tildes */
   if ( ptr = strchr ( report_out, '~' ) )
   {
      /* Found the tilde. Substitute the
         file name for it. */
      *ptr = '';
      sprintf ( msg_buf, "%s%s%s",
          report_out, fn, ptr+1 ) ;

   }
   else
   {
      /* No tilde. Append file name to
         O/S command. */
      sprintf ( msg_buf, "%s %s",
          report_out, fn ) ;
   }

   /* Do the command. */
   system ( msg_buf ) ;

   /* Delete temp file and free its name. */
   remove ( fn ) ;
   free ( fn ) ;
   sm_ferr_reset ( "Command Invoked" ) ;
   break ;

case '':
   /* Message window. Get size of file... */
   size = ftell ( fp ) ;

   /* Allocate memory for it. */
   ptr = malloc ( size + 1 ) ;

   /* Rewind the file */
   fseek ( fp, SEEK_SET, 0 ) ;

   /* Read it into the malloced buffer. */
   fread ( ptr, sizeof ( char ), size, fp ) ;

   /* Close and delete file, free file name */
   fclose ( fp ) ;
   remove ( fn ) ;
   free ( fn ) ;
```

```
      /* null terminate memory buffer of report */
      ptr[size] = '';

      /* Replace all space-newlines with %N */
      for ( ptr1 = ptr ;
          ptr1 = strchr ( ptr1, '' ) ;
          ptr1++ )
      {
         ptr1[-1]='%';
         ptr1[0]='N';
      }

      /* Pop up the message window */
      sm_message_box
         ( ptr, 0, SM_MB_OK|SM_MB_ICONNONE, 0 ) ;

      /* Free up the malloced buffer. */
      free ( ptr ) ;
      break ;

   default :
      /* File appended, just close it. */
      fclose ( fp ) ;
      sm_ferr_reset ( "File appended" ) ;
      break ;
   }
   return ( 0 ) ;
}
```

Status Line

The following example shows how to write a status line function. It is called whenever the logical status line is about to be flushed to the physical display, and ensures that the status line is always printed highlighted and in uppercase.

This function is to be installed as a status line function. The following declarations and definitions, generally found in `funclist.c` or in the main routine source module prepare this routine for installation:

```
/* Include Files */
#include "smdefs.h"    /* screen manager Header File */
#include "smglobs.h"   /* screen manager Globals */

int
statln ( )
{
   int n_columns ;            /* Physical display width */
```

```
    char * stat_text ;          /* Status line text */
    unsigned short * stat_attr;/* Status line attributes */
    int i ;                     /* Loop counter */
    int c;                      /* Upper case stat text char */

    /* Determine width of display */
    n_columns = sm_inquire ( I_MXCOLMS ) ;

    /* Allocate memory for local buffers */
    stat_text = malloc ( n_columns + 1 ) ;
    stat_attr = (short *)calloc ( n_columns,
             sizeof ( short ) ) ;

    /* Copy status text and attributes into buffers */
    strcpy ( stat_text , sm_pinquire ( SP_STATLINE ) ) ;
    memcpy ( ( char * ) stat_attr ,
        sm_pinquire ( SP_STATATTR ) ,
        n_columns * sizeof ( short ) ) ;

     /* Loop through every character on the status line */
    for ( i = 0 ; i < n_columns ; i++ )
    {
       /* Set character to upper case */
       /* Note UNSIGN is defined in smmachs.h to
          remove sign extension */
       c = stat_text [i];
       if ( islower (UNSIGN(c)) )
          c = toupper ( UNSIGN(stat_text[ i ]) ) ;
       stat_text[ i ] = c ;

       /* Add hilight attribute */
       stat_attr[ i ] |= HILIGHT ;
    }

    /* copy local buffer into Panther internal buffers */
    sm_pset ( SP_STATLINE , stat_text ) ;
    sm_pset ( SP_STATATTR , ( char * ) stat_attr ) ;

    /* Free memory */
    free ( stat_text ) ;
    free ( stat_attr ) ;

     return ( 0 ) ;

}
```

CHAPTER

45 Customizing the User Interface

This chapter shows how to set up an interface that is suitable to a GUI platform or user environment. It also discusses strategies for writing applications for non-English-speaking users.

Two distributed files are especially important in designing the user interface:

- The message file contains messages that are seen by end users. It also defines default constants and formats—for example, text labels for message window push buttons, numeric/currency formats, date and time formats, and Panther error messages. You can also use the message file to customize screen editor features for specific languages and format conventions.
- The configuration map file lets you define system-independent aliases for colors, fonts, and line and box styles.

Both files exist independently of the application; this facilitates development and deployment in several ways:

- Definitions are in a single location, so they are easy to access and modify.
- You can edit the source and recompile it independently of the application executables.
- Different parts of an application can access the same definitions, which saves space and development time.
- You can deploy the same executable in different environments, where each distribution uses a different message and configuration map files that suits its own environment. For example, if you need to deploy an application in

different countries, you can create separate message files, which translate message text, date/time formats, and numeric/currency formats to comply with local languages and customs. Similarly, an application that runs on Motif and Windows can use separate configuration files for each platform, which map GUI-specific font and color names to aliases.

Using Message Files

Messages are stored in a binary file that is referenced by the application variable `SMMSGS`, and are loaded into memory at startup. `SMMSGS` can be set in the environment or Windows initialization file, or in a setup file. The Panther configuration directory provides a file of message defaults in source (`msgfile`) and binary (`msgfile.bin`) formats. You can edit the message file source to suit the needs of a given application, then recompile it with the `msg2bin` utility.

Creating and Modifying Message Files

You can edit the source of the distributed message file; you can also supplement this file with message files that contain your own application messages. If you must change the Panther message file, first create a copy of the distributed message file and edit it instead of the original. By doing so, you avoid losing changes with new releases; you also have recourse to the original message file contents.

If you create custom application messages, maintain them in a separate file for these reasons:

- Custom messages are easier to maintain if they are kept in their own file.
- You avoid corrupting Panther messages.
- Custom message files are not overwritten by the new message file that is supplied with each new Panther release.

After you modify an ASCII message file, you must convert it to binary format with `msg2bin`. If the message file includes new entries, you must also create a C header file with `msg2hdr`.

- `msg2bin` converts ASCII message files to binary format for use by Panther. The output of `msg2bin` is a binary file; the utility uses message tags to distinguish between system and user messages. If you have multiple message files, you can compile them into a single binary file by running `msg2bin` with the `-o` option.
- `msg2hdr` converts the message source file to a C header file, which contains #define statements for each user message tag.
- `msg2hdr` can also define JPL global variables, which make the messages accessible in JPL.

How to Create or Add to a Message File

1. Create or access the ASCII message file using a text editor.
2. Edit existing entries or add new entries using the syntax described in the next section. For example:

   ```
   #user messages
   US_NOTAVAIL    = All copies of this movie are unavailable.
   US_ACTL        = Enter an actors last name.
   #administrator messages
   ADM_INVALIDRC = Invalid rating code.
   ```

3. Convert the ASCII file to binary format with the `msg2bin` utility.

   If you only edit the text of existing message tags, your work is done. If you add new entries to the message file, continue with step 4.

4. Define new message tags in a C header file. This makes the messages available to Panther functions such as `sm_fquiet_err`. Do this by running `msg2hdr` on the message source file. Messages are numbered sequentially from 0x0 to 0xF. For example:

   ```
   #define US_NOTAVAIL    0x0
   #define US_ACTL        0x1
   #define ADM_INVALIDRC  0x2
   ```

5. In order to access your messages in JPL, define your tags as global JPL variables in a C header file. Do this by running `msg2hdr` with the -j option on the message source file. For example:

```
global US_NOTAVAIL (1)   0
global US_ACTL (1)       1
global ADM_INVALIDRC (1) 2
```

6. Relink the Panther executable so it includes the new header files (refer to page 42-2).

Note: The message file only contains messages for end users; Panther developer messages are internally defined and cannot be modified.

Message Entry Syntax

Message file entries have the following format:

`tag = msgString`

`tag`

A string that can include letters, digits, and underscores; embedded blanks are invalid. An equal (=) sign must follow *`tag`*. Blanks before and after the equal sign are optional.

If you create your own messages, you can group them according to message tag prefixes. Messages with prefixes can be selectively loaded into memory with `sm_msg_read`. All system messages begin with a standard Panther prefix. These prefixes are reserved and can not be used for messages that you define:

`SM`	Messages and strings used by the Panther runtime library.
`FM`	Messages issued by the screen editor (prodev).
`JM`	Additional runtime messages used by Panther.
`UT`	Messages issued by some Panther utilities.
`DM`	Messages issued by database interactions.

`TP`	Messages issued by the middleware (JetNet and Oracle Tuxedo only).

`msgString`

If *`msgString`* defines a user message, it can contain any alphanumeric string on a single line; the string must contain at least one non-numeric character.

Note: The strings that define formats for date/time and numeric/currency have special syntax requirements.

Refer to page 45-12 for date/time syntax and page 45-19 for numeric/currency syntax.

Leading and trailing spaces are ignored. Enclose embedded and trailing spaces with quotation marks. Panther strips off the quotes when it displays the message.

For example, the distributed message file contains these entries:

```
SM_DAYA6       = Fri
SM_DAYA7       = Sat
SM_MOREDATA    = No more data.
SM_YES         = y
SM_NO          = n
SM_MONL1       = January
SM_MONL2       = February
SM_0DEF_DTIME  = %m/%d/%2y %h:%0M
SM_MB_HELPLABEL= &Help
SM_YN_ERR      = %MuPlease enter %Ky or %Kn into this field.
JM_HITACK      = %MdHit acknowledge key to continue
```

Reserved Characters

The following characters have special usage in message file entries:

`\`	Continues the message string across multiple physical lines in the source file. For example: `PQ_FATALERR = Application unable to post your \` `transaction. Contact your system manager.`
`\n`	Forces a new line.
`&`	Indicates a key mnemonic for push buttons. For example, the following entry lets a user press the letter O on the keyboard instead of choosing the Oui (Yes) acknowledgment push button: `SM_MB_YESLABEL = &Oui`
`#`	If placed in a message file line, comments out that line when `msg2bin` compiles the file.

Note: To use the backslash character in a message, enter two backslashes. *`msgString`* can also contain percent sequences that specify appearance, positioning, and acknowledgment information.

Refer to page 45-8 for information on these.

Missing Entries

If there is no tag for a message or *`msgString`* is missing from the message file and a call is made to display the message, Panther displays the message section and number from the `#define` statement. For example, if the entry for `SM_HITANY` is deleted from the Panther message file, and user input invokes this particular message, the status line displays <8-27>, which is the value for `SM_HITANY` from the include file `smerror.h`.

Message Classes

You can divide messages into message classes. You can define up to eight user classes, numbered 0 to 7. Each class can contain up to 65529 characters. Within each class, you can further differentiate messages through prefixes.

Use section classes and prefixes to divide messages into useful categories. A class of messages or messages of a given prefix can be individually loaded and unloaded from memory through `sm_msg_read` and `sm_msg_del`, respectively. Unclassified messages default to class 0.

Because unclassified messages cannot exceed 65529 characters, an application that requires many messages might require you to divide them into multiple classes.

Note: Panther reserves for its own use classes 8 through 15 and the prefixes described earlier (page 45-4).

Defining a Message Class

Each message class is defined by a message class entry with this format:

`"`*`XY`*`" = `*`digit`*

`"`*`XY`*`"`

A two-character code enclosed in quotation marks. Reserved prefixes (refer to page 45-4) cannot be used.

`digit`

A digit between 0 and 7, inclusive, that designates the class in library functions such as `sm_msg_read`.

All entries below a message class entry are part of the same message class. For example, the following message file excerpt defines two message classes with prefixes `U0` and `U1`:

```
"U0"= 0
U0_BADVAL    =  Bad value
U0_WRONGDATE =  Date must be with 30 days of current date
...

"U1" = 1
WRONGRATE =  This is not the applicable rate.
...
```

Given these entries, you can issue a command that reads all messages in class 0 that begin with the prefix `U0`:

```
sm_msg_read ("U0", 0, MSG_FILENAME|MSG_NOREPLACE, "umsg.bin")
```

This command reads all messages in class 0, irrespective of prefixes:

```
sm_msg_read ("", 1, MSG_FILENAME|MSG_NOREPLACE, "umsg.bin")
```

When the message file is compiled with `msg2bin`, tags are used to distinguish between system and user messages. User-defined messages are numbered consecutively, starting with the class number times `0x1000`. Unclassified messages are numbered from zero. For example, the fourth message in user class four is numbered `0x4004`. As a developer, you must remember to maintain the order of user messages and the assignment of their identifiers.

Setting Message Display and Behavior Options

Several percent escape options let you control message content and presentation. The character or characters that follow the percent sign are case-sensitive; type them exactly as shown. This prevents conflicts with percent escape options used by `printf` and the tokens used by date/time formats. Some percent escape options must appear at the beginning of the message; others are valid only for display in a window or on the status line.

Table 45-1 summarizes the available escape sequences, followed by detailed information about each option.

Table 45-1 Percent escape options for messages

Option	Description
`%Ahhhh`	Change display attributes. Valid for status line messages only.
`%K`	Display key label.
`%B`	Beep the terminal. This option must precede the message text.
`%N`	Use a carriage return in the message text. Forces message to display in popup window.
`%W`	Display message in a popup window.
`%Md`	Force the user to press the acknowledgment key (`ER_ACK_KEY`) in order to dismiss the error message. 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.

Table 45-1 Percent escape options for messages

Option	Description
`%Mu`	Force message display to the status line and permit any key board or mouse input to serve as acknowledgment. Panther then processes the keyboard or mouse input. This option must precede the message text.

`%A` *`attr-value`*—Change display attributes

Valid only for status line messages, you can place `%A`*`attr-value`* anywhere in the message text. It changes the display attributes of the text that follows it. *`attr-value`* is a four-digit hexadecimal value that represents one display attribute or the sum of two or more attributes.

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 for valid attribute values.

Note: Monochrome terminals ignore color attributes. However, if you are developing for color terminals, include a color code with the %A. Otherwise, both the foreground and background colors default to black when the %A is not followed by a color code.

Table 45-2 lists the display attributes and their hexadecimal codes as defined in the include file smattrib.h.

Table 45-2 Display attributes and hexadecimal codes for status line messages

Foreground Attributes*		**Background Attributes**	
Attribute Mnemonic	Hex Code	Attribute Mnemonic	Hex Code
`REVERSE`	`0010`	`B_HILIGHT`	`8000`
`UNDERLN`	`0020`		
`BLINK`	`0040`		
`HILIGHT`	`0080`		

Table 45-2 Display attributes and hexadecimal codes for status line messages

Foreground Attributes*		Background Attributes	
DIM	1000		
Foreground Colors		Background Colors	
BLACK (colors are additive)	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 (inherit color from container)	4000

**Attributes are additive. One or more foreground attributes can be added to a background attribute, foreground color and background color.*

For example, the following message appears in red characters on the default black background with Warning. in blinking characters.

```
SM_WARNBIG= %A44Warning.\
%A0004Form is larger than screen size.
```

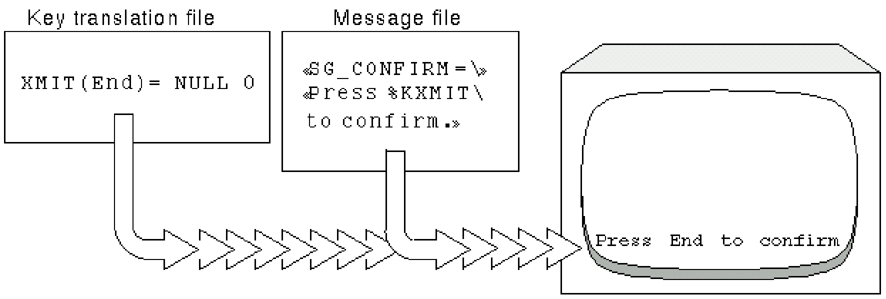
%B — Beep terminal

Place %B in a status line or message so that the terminal beeps via sm_bel when the message is displayed. This option must precede the message text.

%K—Display key label

Place %Klogical-key anywhere in the text of status line and error messages. Panther interprets logical-key as a mnemonic defined in smkeys.h. If the key translation file defines a key label for the logical key, the key label replaces the percent sequence in the message text. If there is no key label or no such logical key, `%K` is stripped off and logical-key remains in the message text.

Refer to page 6-2 in the *Configuration Guide* for more information about key translation files.



```
XMIT(End)= NULL 0
```

```
«SG_CONFIRM=\»
«Press %KXMIT\
to confirm.»
```

```
Press End to confirm
```

Figure 45-1 The key label is defined in the key translation file; the message file contains the message text. The result is displayed on the screen.

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 user to acknowledge error message

Place `%Md` at the beginning of an error message so that the user is forced to press the predefined acknowledgment key `ER_ACK_KEY` to clear the message. If the user presses any other key, Panther displays an error message or beeps, depending on how application variable `ER_SP_WIND` is set. The keypress is not processed as data.

The `%Md` option corresponds to the default message behavior when application variable `ER_KEYUSE` is set to `ER_NO_USE`. If `ER_KEYUSE` is set to `ER_USE`—that is, your application default does not require use of the acknowledgement key—set `%Md` in a message in order to force the user to press the acknowledgment key to clear a message.

`%M`*`[time-out]`*—Display transient error message

Place `%Mt` at the beginning a transient status line message. 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:

```
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. When the message is displayed in a window, users dismiss the message by choosing OK or by pressing the acknowledgement key; and Panther discards any keyboard input.

`%Mu`—Use any key to acknowledge error message

Valid only for error messages, you must place `%Mu` at the beginning of an error message. Panther forces message display to the status line and permits any keyboard or mouse input to serve as error acknowledgment. Panther then processes the keyboard or mouse input. In the following example, entering y or n acts as both message acknowledgment and data entry:

```
%MuPlease enter %Ky or %Kn into this field.
```

When the message is displayed in a window, users dismiss the message by choosing OK or pressing the acknowledgement key. Panther then discards any keyboard input.

`%N`—Insert line returns in message text

Insert one or more `%N` options in a message to force line returns in a windowed message. By default, message text wraps within the window.

`%W`—Display message in a window

Valid only for error messages, forces display in a window. Place %W at the beginning of the message.

Customizing Date and Time Formats

The Panther message file includes entries that establish date and time formats and text. It also includes substitution variables that the screen editor displays as options for a date/time widget's Format Type property.

Modify date/time entries in the message file for these reasons:

- Translate the text (days of the week and names of months) to comply with local customs.
- Customize the formats (`SM_` date/time entries) to comply with local customs or individual preferences.
- Customize the names of format mnemonics and format types for non-English developers or for individual preferences.

This section describes the tags and their default entries and how you can change the entries to meet the needs of both your development environment and your application.

Date/Time Defaults

When you choose Date/Time for the Data Formatting property of a widget, ten default choices for Format Type are available. The message file defines these format types: a name tag defines the name of a format type to appear on the Format Type option menu; and a corresponding format tag specifies the format associated with that name. For example `FM_3MN_DEF_DT` defines the name of the first format type as `MON/DATE/YR4 HR:MIN2` and the corresponding format tag `SM_3DEF_DTIME` defines its format as `%m/%d/%4y%h:%0M`.

Table 45-3 lists the date/time name tags as delivered with Panther and their corresponding format type names, listed as they appear in the Properties window. The entries in Table 45-4 define the formats that correspond to the date/time tags and names in Table 45-3. (The tokens in the formats are defined in Table 45-5.)

Table 45-3 Default date/time entries

Date/Time tag	Format type	Formatting result
`FM_3MN_DEF_DT`	`MON/DATE/YR4 HR:MIN2`	`4/1/2016 13:13`
`FM_4MN_DEF_DT`	`MON/DATE/YR4`	`4/1/2016`
`FM_0MN_DEF_DT`	`MON/DATE/YR2 HR:MIN2`	`4/1/16 13:13`
`FM_1MN_DEF_DT`	`MON/DATE/YR2`	`4/1/16`
`FM_2MN_DEF_DT`	`DEFAULT TIME`	`13:13`

Tags FM_5MN_DEF_DT through FM_9MN_DEF_DT are undefined in the provided message file; The Format Type property displays them as DEFAULT5 through DEFAULT9; they have the same format specification as FM_5MN_DEF_DT. The corresponding formats are defined in the message file, as shown in Table 45-4.

Table 45-4 Default date/time formats

Date/Time format tag	Tokenized format
SM_0DEF_DTIME	%m/%d/%2y %h:%0M
SM_1DEF_DTIME	%m/%d/%2y
SM_2DEF_DTIME	%h:%0M
SM_3DEF_DTIME	%m/%d/%4y %h:%0M
SM_4DEF_DTIME	%m/%d/%4y
SM_5DEF_DTIME	%m/%d/%2y %h:%0M
...	...
SM_9DEF_DTIME	%m/%d/%2y %h:%0M

Date/Time Tokens

When specifying a format in the message file or as an argument to the library functions sm_sdtime or sm_udtime, you must use some combination of tokens—not those in the Properties window (MON or DEFAULT5). In this way, Panther does not need to parse the message file, and the library functions can be used without knowing the names of substitution variables defined in the message file. When Panther performs date calculations using a format, it replaces tokens with their appropriate values. All other characters in the format such as, commas, slashes, and colons are used literally. If you wish to refer to one of the default format types, there are format tokens ranging from %0f to %9f that correspond to each of the format tags (SM_ date/time entries).

The tokens are listed in Table 45-5. Most of these substitute a numeric value; message entries are indicated for those that substitute text. For example, %4y might substitute 1999, whereas %*m would, depending on the date, substitute one of the values defined by SM_MONL1 through SM_MONL12, perhaps July, perhaps Juillet.

Table 45-5 Definitions of date and time tokens

Description	Token	Message entries for text
Year:		
4 digit	`%4y`	
2 digit	`%2y` (Use setup file to specify century break)	
Month:		
numeric (1 or 2 digit)	`%m`	
numeric (2 digit)	`%0m`	
abbreviated name (3 char)	`%3m`	`SM_MONA1...SM_MONA12`
full name	`%*m`	`SM_MONL1...SM_MONL12`
Day of the month:		
numeric (1 or 2 digit)	`%d`	
numeric (2 digit)	`%0d`	
Day of the week:		
abbreviated name (3 char)	`%3d`	`SM_DAYA1...SM_DAYA7`
full name	`%*d`	`SM_DAYL1...SM_DAYL7`
numeric	`%.d`	
Day of the year:		
numeric (1-366)	`%+d`	
Time:		
hour (1 or 2 digit)	`%h`	

Table 45-5 Definitions of date and time tokens

Description	Token	Message entries for text
hour (2 digit)	`%0h`	
minute (1 or 2 digit)	`%M`	
minute (2 digit)	`%0M`	
second (1 or 2 digit)	`%s`	
second (2 digit)	`%0s`	
AM and PM	`%p`	`SM_AM, SM_PM`
Ten default formats:		
7. formats specified in message file entries*	8. `%0f - %9f`	9. `SM_0DEF_DTIME to SM_9DEF_DTIME`
10. Other:		
11. literal percent sign	12. `%%`	
13. *Tokens provided so default formats can be used with library functions `sm_sdtime` and `sm_udtime`.		

Creating Date and Time Defaults

Ten date and time entries are available to define formats and the names that specify them. The tokens for `SM_5DEF_DTIME` through `SM_9DEF_DTIME` are defined to have the same format as `SM_0DEF_DTIME`. You can use these additional tags to create and name your own date/time formats.

How to Change or Create a Default Date/Time Format

1. Open the ASCII version of the message file with a text editor.

2. Change one of the `SM_` date/time entries (`SM_0DEF_DTIME` through `SM_9DEF_DTIME`) to define the desired format.

   For example, the `DEFAULT5` substitution variable has this initial format:

   ```
   SM_5DEF_DTIME =  %m/%d/%4y %h:%M0
   ```

   `You can change this entry as follows:`

   ```
   SM_5DEF_DTIME =  %d %*m %4y %h:%M0 %p
   ```

   Widgets that have their date Format Type property set to `DEFAULT5` display dates in the format:

   ```
   30 November 2016 3:40 PM
   ```

3. Create the substitution variable with the corresponding `FM_` tag:

   ```
   FM_5MN_DEF_DT = DATE and TIME
   ```

   `DATE and TIME appears as an option for the Format Type property.`

   A widget whose Format Type is set to `DATE` and `TIME` displays dates in this format:

   ```
   30 November 2016 3:40 PM
   ```

4. Convert the ASCII message file to binary format with the `msg2bin` utility.

Note: Tokens are provided (refer to Table 45-5 on page 45-15) that correspond to each of the default formats so that these defaults can be used with the library functions `sm_sdtime` and `sm_udtime`.

Defaults for Non-English Applications

To customize the date and time entries in the Panther message file for non-English applications, you can:

- Translate the text entries which name the days of the week and the months of the year. This text is assigned to the tags `SM_DAYA1` ... `SM_DAYA7` (abbreviated names of days), `SM_DAYL1` ... `SM_DAYL7` (full names of days), `SM_MONA1` ... `SM_MONA12` (abbreviated names of months), `SM_MONL1` ... `SM_MONL12` (full names of months).
- Change the formats associated with the `SM_` date/time tags to comply with local customs.

By translating text and changing formats, widgets using the Format Type specification described in the example shown earlier on page 45-16, `DEFAULT5` might appear as:

```
30 Novembre 2016  3:40 PM
```

To develop an application for French users, translate the text assigned to `SM_DAYA1...SM_DAYA7, SM_DAYL1..SM_DAYL7, SM_MONA1...SM_MONA12, and SM_MONL1...SM_MONL12`, like this:

```
SM_DAYA1  =  Dim
SM_DAYA2  =  Lun
SM_DAYA3  =  Mar
...

SM_DAYL3  =  Mardi
SM_DAYL4  =  Mercredi
SM_DAYL5  =  Jeudi
SM_DAYL6  =  Vendredi
SM_DAYL7  =  Samedi
...

...

SM_MONA1  =  Jan
SM_MONA2  =  Fév
SM_MONA3  =  Mar
...
SM_MONL7  =  Juillet
SM_MONL8  =  Août
SM_MONL9  =  Septembre
SM_MONL10 =  Octobre
SM_MONL11 =  Novembre
SM_MONL12 =  Décembre
```

This method can be useful if you are distributing the same application to users who speak different languages. A user's `SMMSGS` variable in the local setup (`smvars`) file or system environment can specify the name of the appropriate message file and screen libraries. Date/time fields then display the date in a language and format familiar to the user, while all programming code remains independent of the user's language.

Translating Defaults for Developers

In addition to translating the text for days of the week and months of the year, and localizing formats, you can translate the names of substitution variables for non-English speaking developers, These entries are adjacent to each other in the Panther message file, beginning with `FM_YR4` and ending with `FM_9MN_DEF_DT`.

For example, you might provide these entries for French-speaking developers:

```
FM_YR4   =  ANNÉE4
```

```
FM_YR2  =  ANNÉE2

FM_MON  =  MOIS

FM_MON2 =  MOIS2

FM_DATE =  JOUR

...
```

Given these changes, French-speaking developers using Panther can create date/time formats using substitution variables in their native language—`MOIS2`, `ANNÉE4`, and `JOURA`, while Spanish-speaking developers might use substitution variables like `MES2`, `AÑO4`, and `DÍAA`.

Literal Dates in Calculations

The Panther message file includes an entry to specify the format of literal dates used in `@date` calculations. The tag `SM_CALC_DATE` specifies the format. The default format is `%m/%d/%4y` (`MON/DATE/YR4`). For example, to count the number of days until the millennium, the library function `sm_calc` can be used with a literal date:

```
sm_calc (0,0,'days = @date(1/1/2000)- @date(today)');
```

Numeric Formats

When you choose Numeric for the Data Formatting (`data_formatting`) property of a widget, ten default choices for Format Type (`numeric_type`) are made available to you. The message file contains the definitions for these format types: A name tag defines the name of a format type to appear for the Format Type property in the Properties window, and a corresponding format tag specifies the format associated with that name. For example `SM_0MN_CURR_DEF` defines the name of the first format type as Local Currency and the corresponding format tag `SM_0DEF_CURR` defines its format as "`.22,l$`".

You can modify the message file to store ten different default numeric formats. Like the date/time message entries, one entry (`SM_0DEF_CURR` through `SM_9DEF_CURR`) defines the format, and a corresponding entry (`SM_0MN_CURRDEF` through `SM_9MN_CURRDEF`) specifies what the screen editor displays as options for a numeric widget's Format Type (`numeric_type`) property.

Numeric Format Syntax

Numeric formats are defined as `r m x t p c c c c c`:

`r`	Radix separator or decimal symbol (usually a period or comma)
`m`	Minimum number of decimal places
`x`	Maximum number of decimal places
`t`	Thousands' separator (i.e., a comma or period; `b` for a blank; or `n` to not use a thousands' separator)
`p`	Placement of currency symbol (`l` = left, `r` = right, or `m` = middle), or omit to not use a currency symbol
`ccccc`	Currency symbol (up to 5 characters, including blank spaces)

To specify leading or trailing blanks in a format, enter blank spaces before or after the currency characters. The spaces become a part of the currency symbol.

For example, the format `.22,l$` contains these specifications:

- period (decimal point) as the radix separator
- minimum of two decimal places
- maximum of two decimal places
- comma as the thousands separator
- the currency symbol precedes (to the left of) the number.
- the dollar sign (`$`) is the currency symbol

Formats in Provided Message File

Table 45-6 lists the numeric tags as delivered with Panther, their format type name, and the corresponding format tag and the default format. Description names are defined only for the first three format types. (Names for format types default to `default'). The last seven names and formats are for other types you can custom define. Therefore, the last seven format types are defined identically to `SM1_DEF_CURR`, which is set to two decimal places with a comma as the thousands separator.

Table 45-6 Default message entries for defining numeric formats

Numeric tag	Format type name	Corresponding format tag	Default format
`SM_0MN_CURRDEF`	Local Currency	`SM_0DEF_CURR`	`".22,l$"`
`SM_1MN_CURRDEF`	2 decimal places with commas	`SM_1DEF_CURR`	`".22,"`
`SM_2MN_CURRDEF`	0 decimal places with commas	`SM_2DEF_CURR`	`".00,"`
`SM_3MN_CURRDEF`		`SM_3DEF_CURR`	`".22,"`
`SM_4MN_CURRDEF`		`SM_4DEF_CURR`	`".22,"`
		`...`	`...`
`SM_9MN_CURRDEF`		`SM_9DEF_CURR`	`".22,"`

`SM_0MN_CURRDEF` defines the name of the format type as Local Currency, and the corresponding format tag `SM_0DEF_CURR` defines its format as `".22,l$"`. A widget with this property specification would have data formatted, for example, as `$5,100.75`.

Creating a Default Numeric Format

A message file can specify ten numeric format entries. You can change any or all formats to suit your application's requirements. To create a numeric format:

- Edit the numeric format associated with a format tag.
- Define its corresponding numeric tag to equal the name of your newly created format variable.

How to Customize a Default Numeric Format

1. Open the ASCII version of the message file with a text editor.

2. Change one of the `SM_` numeric entries (`SM_0DEF_CURR` through `SM_9DEF_CURR`) to define the desired format.

   To specify leading or trailing blanks in a format, enter blank spaces before or after the currency character. The spaces become a part of the currency symbol.

   For example, you can add a format for the French franc with this change:

   ```
   SM_9DEF_CURR = ',22.r F'
   ```

3. Add a descriptive definition in the corresponding `SM_` numeric entry. For example:

   ```
       SM_9MN_CURRDEF = Franc
   ```

   The Format Type property in the Properties window will display Franc as one of the values you can assign to a widget with numeric data.

4. Convert the ASCII message file to binary format with the `msg2bin` utility.

   Given the previous definition, the Format Type property in the Properties window displays Franc as one of the values you can assign to widgets with numeric data. Widgets thus formatted show currency data in the form:

   ```
   999.999,99 F.
   ```

Translating Defaults for Developers

In addition to modifying the numeric formats to comply with local customs, you can translate the names that appear on the numeric format type property menu for non-English speaking developers. The first three entries are adjacent to each other in the Panther message file, beginning with `SM_0MN_CURRDEF` and ending with `SM_3MN_CURRDEF`.. `SM_4MN_CURRDEF` through `SM_9MN_CURRDEF` are also available variables but not predefined in the message file.

For example, you might provide these entries for Spanish-speaking developers:

```
SM_0MN_CURRDEF = DINERO

SM_1MN_CURRDEF = NUMERO
```

Given these changes, Spanish-speaking developers see format type choices in their own language.

Decimal Symbols

Via the message file tag `SM_DECIMAL` you can define a default decimal symbol (or radix separator). When you define a widget to have or accept numeric data, you can also specify any decimal symbol (or radix separator). However, the `SM_DECIMAL` entry enforces a default symbol. If `SM_DECIMAL` is not specified in the message file, Panther tries to determine the appropriate symbol from the operating system.

Panther accommodates three types of decimal symbols. These decimals differ in scope and function.

system

The character that is used by the operating system when translating characters to internal values or vise versa—for example, in C functions `atof` and `sprintf`. The default is period.

Note: Setting the system decimal symbol incorrectly can cause unexpected results when Panther processes numeric values.

local

Defined by the message file entry for `SM_DECIMAL`, by default set to period. This setting overrides the system symbol within a Panther application. Set `SM_DECIMAL` according to local customs—for example, period in English-speaking countries; comma in Europe. If the system and local symbols are different, Panther translates appropriately when interacting with system routines.

widget

Set for a specific widget through its `decimal_symbol` property (refer to on page 10-20 in the *Using the Editors)*. This symbol is used only for data entry validation and for displaying widget values. Use widget-level decimal symbols when you need to handle multiple decimal conventions within a single application.

Customizing Push Button Labels for Message Boxes

The message file tags `SM_MB_OKLABEL` through `SM_MB_HELPLABEL` provide the text for message box push buttons.

Note: Microsoft Windows for other languages automatically translates standard push buttons to the appropriate language.

How to Change/Translate Push Button Labels

1. Access the ASCII version of the message file with a text editor.
2. Change the label text. Place an ampersand before the character to serve as the push button's key mnemonic.
3. Convert the ASCII message file to binary format with `msg2bin`.

For example, if you were converting your application to Spanish, you might include the following in your message file:

```
SM_MB_OKLABEL       =  &Ok
SM_MB_CANCELLABEL   =  &Cancelar
SM_MB_YESLABEL      =  &Si
SM_MB_NOLABEL       =  &No
SM_MB_RETRYLABEL    =  &Re-intentar
SM_MB_IGNORELABEL   =  &Ignorar
SM_MB_ABORTLABEL    =  A&bortar
SM_MB_HELPLABEL     =  &Ayuda
```

Setting Yes/No Values

The values associated with the message tags `SM_YES` and `SM_NO` can be translated or standardized to meet your development or application's requirements. For example, you can translate the value for `SM_YES` to s (short for sí) for Spanish-speaking users.

Library functions such as `sm_is_yes`, and properties such as keystroke filter that use the `SM_YES` and `SM_NO` entries expect and return the appropriate character as defined in the message file. In the case of a Spanish-speaking user, entering s (for an affirmative response) is recognized, whereas y is ignored.

Using Alternate Message Files

The `SMMSGS` application variable specifies the binary file to read into memory at Panther's initialization. If you serve an international market, you can give users the option of selecting from alternate message files. At runtime the user can set the `SMMSGS` for the binary message file that is appropriate to his/her language.

Alternative files for an application (and for non-English versions of Panther) must be identical in terms of the number and sequencing of all messages (refer to page 45-4 for information about adding messages).

Configuration Map File

The configuration map file contains definitions for screens and widgets—colors, fonts, lines and box styles—that you can tailor to different platforms. The file is divided into several sections:

- `[Colors]` maps user-defined color names to system color names (page 45-26).
- `[Schemes]` maps color definitions to application components such as screens and widget types (page 45-30).
- `[Lines]` defines line and box styles (page 45-34).
- Several font sections that tell Panther which fonts and font sizes to display in the drop-down menus in the screen editor; they also define default fonts for display and report output, and map system-specific font names to Panther font aliases (page 45-36).

By defining these elements in GUI-specific files and using their names for screen and widget properties, you can create applications that are easy to port across different platforms. Instead of creating multiple instances of the same screens or reports that use GUI-specific font and color names, you can create multiple configuration map files—one for each platform on which your applications run. For example, you can create a color alias PanicButtonRed that resolves to different colors in different configuration maps.

Panther is installed with at least one configuration map file (*`cmap`) that suits your environment. You can edit these or create your own with an ASCII text editor, then run the utility `cmap2bin` to convert it to binary format.

During initialization, Panther looks for the application variable `SMCOLMAP` which can be defined in the environment or in an `SMVARS` file. This variable gives the full path name of the binary configuration map file.

Defining Colors

When you create a screen or widget, the screen editor seeks default color settings, or a scheme, for that object's type, foreground and background. The editor automatically sets the color property to Scheme and then resolves the scheme, looking first in the configuration map file. If the file provides no scheme for the object, the editor looks elsewhere for color defaults (refer to page 45-30).

Panther provides configuration map files with default schemes for screens and for each widget type. You can define your own color schemes that suit your environment, style preferences, or development and application requirements. Or you can rely on the local GUI to assign colors to your application objects.

You can set the Color Type property to one of these three settings:

- Scheme—The defaults defined in the `[Schemes]` section of the configuration map file, or if none, a set of default colors determined by settings defined either in the native GUI or in Panther.
- Basic—Panther's eight highlighted and eight unhighlighted colors, plus the Container option. (The Container option specifies that a widget within another object has the same background color as the container.)
- Extended colors—GUI-specific colors that are specified by a string and are resolved in the `[Colors]` section of the configuration map file, or directly by the GUI.

Defining Color Aliases

The `[Colors]` section defines GUI-independent color aliases that you can use in the Color Name property of screens and widgets. All color names, including Panther palette color names like `hilight_red`, must be added to the list of color aliases. Each entry appears on its own line in the following format: `aliasColor = color`

`aliasColor`
: Any name you choose to identify a color.

`color`
: One of the following:

  - An RGB value in a platform-specific form:

For Windows, use the form "red/green/blue" where red, green and blue are numbers between 0 and 255. For example:

```
PanicButtonRed = "205/92/92"
```

For Motif, use the form "#RedGreenBlue" where Red, Green, and Blue are hex numbers between `00` and `ff`. For example:

```
PanicButtonRed = "#cd5c5c"
```

- A GUI specific name (in Motif only). For more information on Motif color names, refer to page 4-5.in the *Configuration Guide.* For example:

  ```
  PanicButtonRed = "Indian Red"
  ```

- Panther keywords in the form basicColor (attributes); where basicColor corresponds to one of Panther's 16 colors (or “container”) and attributes is an optional display attribute (refer to Table 45-7). For example:

  ```
  NumberField = BLACK UNDERLN
  ```

  This style of definition can create a GUI-independent color alias. For example:

  ```
  SpringGreen=Green Hilight
  SummerGreen=Green Dim
  ```

  If a widget had the Motif color SpringGreen specified and Panther could not find it, it would substitute the Panther color Green Hilight, which is always defined. A configuration map with similar aliases would allow a Motif-specific screen to appear similarly when running in Panther's character mode.

Note: Under Windows, Panther screens and widgets that have highlighted background colors are different from those having unhighlighted background colors. Panther display attributes have no effect in Motif.

Table 45-7 Panther color and attribute keywords

Color	Color keyword	Attribute	Attribute keyword
Black	`BLACK`	Reverse video	`REVERSE`
Blue	`BLUE`	Underline	`UNDERLN`

Table 45-7 Panther color and attribute keywords

Color	Color keyword	Attribute	Attribute keyword
Green	`GREEN`	Blink	`BLINK`
Cyan	`CYAN`	Highlight	`HILIGHT`
Red	`RED`	Dim	`DIM`
Magenta	`MAGENTA`		
Yellow	`YELLOW`		
White	`WHITE`		
Container	`CONTAINER`		

The keyword `CONTAINER` specifies that a widget within another object has the same background color as the container. Therefore `CONTAINER` cannot be used to specify a foreground color. Also, because `CONTAINER` may contain attributes, you cannot specify any additional attributes with it.

Editor Colors

A number of predefined color aliases control the editor's appearance. All editor color aliases begin with se; entries use the same format as user-defined colors. For example, this entry in a Windows configuration map file sets the background color of the design screen:

```
seFormBg = "127/255/0"
```

Table 45-8 lists screen editor color aliases and the objects whose appearance they control:

Table 45-8 Screen editor object constants (object keywords are case sensitive)

Color alias	Description
`seBorderFG`	Editor windows border foreground.
`seCheckFG`	Property window option menu foreground.

Table 45-8 Screen editor object constants (object keywords are case sensitive)

Color alias	Description
`seEntryFG`	Property window text field foreground.
`seFormBG`	Editor windows background.
`seLabelFG`	Label foreground.
`seListBG`	List background (except for Property window).
`seListFG`	List foreground (except for Property window).
`seMultiBG`	Multiline text background.
`seMultiFG`	Multiline text foreground.
`seOptionmenuBG`	Option menu background.
`sePushBG`	Push button background.
`sePushFG`	Push button foreground.
`sePwListBG`	Property window list background.
`sePwListFG`	Property window list foreground.
`seTbBorderFG`	Tool box border foreground.
`seTbFormBG`	Tool box background.
`seTbTogFG`	Tool box toggle button foreground.
`seTextBG`	Text background.

Sample Colors Section

The following examples are from ASCII configuration map files; the aliases ensure that colors specified for one platform display correctly on others without editing Color Name properties of application components. Given the appropriate configuration map file, an application displays colors that are correct for its environment.

The `[Colors]` section in the Motif configuration map defines these color aliases:

```
Slate Gray   = "#708090"
Olive Drab   = "#6B8E23"
ButtonBlue   = "#0938EE"
```

For character mode, the `[Colors]` section redefines these aliases with Panther color names:

```
Slate Gray   = HILIGHT WHITE
Olive Drab   = GREEN
ButtonBlue   = BLUE
```

For a Windows configuration map, these aliases are redefined with RGB values:

```
Slate Gray   = "112/128/144"
Olive Drab   = "107/142/35"
ButtonBlue   = "09/38/240"
```

If you specify Slate Gray on the three platforms, the correct color is displayed. If you alias the Motif color to map to a Panther-specific color, you ensure that when your application runs in character-mode Panther, Slate Gray is displayed as the Panther color hilight white.

Defining Color Schemes

You can decide on a set of default colors for each newly created object in the screen editor. When the Color Type property is set to Scheme, Panther uses the configuration map file to resolve the object's foreground and background colors, according to its type.

The Schemes section of the configuration map file can include explicit settings or defer to the GUIs resource database or initialization file.

Default Schemes

If the configuration map file omits a `[Schemes]` section, Panther uses the following default schemes:

- Windows: Control Panel colors.
- Motif: the *fg and *bg settings in the resource database.
- Character-mode Panther: white foreground, black background.

Scheme Syntax

Each entry in the `[Schemes]` section appears on its own line in the following format:

`object = color`

`object`

Any widget type, including lines and boxes, screen, and borders, followed by either a foreground (FG) or background (BG) mnemonic; for example, `ToggleButtonFG` and `ListBoxBG`. Refer to Table 45-9 for a list of valid object specifications.

`color`

One of the following specifications:

- An RGB value in a platform-specific form:

  For Windows, use the form "red/green/blue" where red, green and blue are numbers between 0 and 255. For example:

  ```
  MultiTextFG = "0/0/255"
  ```

  For Motif, use the form "#RedGreenBlue" where Red, Green, and Blue are hex numbers between `00` and `ff`. For example:

  ```
  MultiTextFG = "#0000ff"
  ```

- Panther keywords in the form `basic_color` [*attribute*]; where `basic_color` corresponds to one of Panther's 16 colors (or "container") and attribute is an optional display attribute. (Refer to Table 45-7.) You may not use the container color for foreground color designations. For example:

  ```
  TEXTFG = BLACK UNDERLN
  ```

- A GUI independent color alias. For example, this entry sets `LabelBg` to Panther blue, an alias that must be defined in the [Colors] section:

  ```
  LabelBG=Panther blue
  ```

- Use the keyword GUI to indicate the native GUI resource database or initialization file. For example, the following indicates that the native GUI resolves the foreground color for toggle buttons:

  ```
  TogglebuttonFG=GUI
  ```

- Use the keyword GUI and the Motif or Windows screen element scheme value. (Do not use any additional attributes with a GUI keyword color designation.) For example, for Windows:

  ```
  PushButtonFG = GUI Buttonface
  ```

  For Motif:

  ```
  PushButtonBG = Prolifics*background
  ```

- GUI-specific colors. These exist only in Motif; their usage limits the scheme's color portability to other environments. For more information on Motif color names, refer to page 4-5 in the *Configuration Guide*.

  ```
  CheckBoxFg =tomato
  ```

Table 45-9 Objects for setting schemes (keywords are case-insensitive).

Object Specification	Descriptions
`BoxTopBg`	Top border background of box widget.
`BoxTopFg`	Top border foreground of box widget.
`CardBg`	Tab card widget background (ignored in Windows).
`CardFg`	Tab card widget foreground (ignored in Windows).
`CheckBoxBg`	Check box background.
`CheckBoxFg`	Check box foreground.
`ComboBoxBg`	Combo box background.
`ComboBoxFg`	Combo box foreground.
`DeckBg`	Tab deck widget background (ignored in Windows).
`FormBg`	Screen color scheme.
`FormBorderBg`	Screen border background.
`FormBorderFg`	Screen border foreground.
`GraphBg`	Graph widget background.
`GraphFg`	Graph widget foreground.

Table 45-9 Objects for setting schemes (keywords are case-insensitive).

Object Specification	Descriptions
`GridBg`	Grid widget background.
`GridFg`	Grid widget foreground.
`LabelBg`	Static label background.
`LabelFg`	Static label foreground.
`LineBg`	Line widget background.
`LineFg`	Line widget foreground.
`ListBoxBg`	List box background.
`ListBoxFg`	List box foreground.
`MultiTextBg`	Multitext background.
`MultiTextFg`	Multitext foreground.
`OptionMenuBg`	Option menu background.
`OptionMenuFg`	Option menu foreground.
`OutputBg`	Dynamic label background.
`OutputFg`	Dynamic label foreground.
`PushButtonBg`	Push button background.
`PushButtonFg`	Push button foreground.
`RadioButtonBg`	Radio button background.
`RadioButtonFg`	Radio button foreground.
`ScaleBg`	Scale widget background.
`ScaleFg`	Scale widget foreground.
`TextBg`	Single line text background.
`TextFg`	Single line text foreground.

Table 45-9 Objects for setting schemes (keywords are case-insensitive).

Object Specification	Descriptions
`ToggleButtonBg`	Toggle button background.
`ToggleButtonFg`	Toggle button foreground.

Defining Line and Box Styles

[Lines] section entries map character-mode styles for lines and boxes to GUI styles. Character-mode line and box styles are defined in the box and border entries of your terminal's video file.

Each entry appears on its own line in the following format:

`styleName = styleContent`

`styleName`
A predefined or new style name. Spaces are allowed and case is irrelevant.

`styleContent`
A predefined style name or another alias style name defined in this file. Spaces are allowed and case is irrelevant. Currently supported predefined style names include:

Dash	Dashdot	Dashdotdot	Default
Dot	Double Dash	Double	Etched In
Etched In Dash	Etched Out	Etched Out Dash	In
None	Out	Single	
Style 0	Style 1	...	Style 9

You can use this section of the configuration map file to assign the styles 0 through 9 to GUI-specific line styles. For example, you might define the following entries for Motif:

```
[Lines]

style 0 = etched in
style 1 = etched out
```

These entries tell Panther for Motif that when to interpret `style 0` as an alias for etched in, and `style 1` as etched out.

Character Mode Styles

Styles 0 through style 9 are native to Panther running in character mode. Style 1 is defined as the default line style. When you assign a character-specific style as the Style property value for a line or box style in the screen editor, that style is mapped to style 1 on non-character Panther applications. GUI-specific styles map to style 1 when running in character mode.

GUI Styles

The default line and box style for all GUI platforms is etched in. Table 45-10 shows which styles are supported by different platforms, and how Panther displays styles that are undefined or are not supported by the GUI. Supported styles are represented by asterisks (*). Because Windows supports the same styles for lines and boxes, the table does not differentiate between these two widgets; however, Motif supports a different set of styles for each widget type, so these are depicted separately.

Note: Under Windows, screens that have their 3D property set to No display Etched In and Etched Out as single lines.

Table 45-10 Mapping of Panther line and box styles on GUI platforms

Line styles	Windows	Motif line	Motif box
Default	etched in	etched in	etched in
Style 0	single	no line	etched in
None	single	no line	etched in
Styles 1-9	single	etched in	etched in
Etched In	*	*	*
Etched In Dash	dash	*	etched in

Table 45-10 Mapping of Panther line and box styles on GUI platforms

Line styles	Windows	Motif line	Motif box
Etched Out	*	*	*
Etched Out Dash	dash	*	etched in
Single	*	*	etched in
Dash	*	*	etched in
Dot	*	dash	etched in
Dashdot	*	dash	etched in
Dashdotdot	*	dash	etched in
In	single	etched in	*
Out	single	etched out	*
Double	single	*	etched in
Double Dash	single	*	etched in

** Style is supported by the GUI platform.*

To control mapping, assign the desired specification in the configuration map file.

Setting Display and Printing Fonts

Display font defaults and aliases are defined in a single section—`[Windows Fonts]` for Windows and `[Display Fonts]` for other platforms.

Font defaults and aliases for printed reports are defined in one of four sections: in the `[Windows Fonts]` font display section for output from Windows print drivers, a `[Postscript Fonts]` section for PostScript and PDF output, a `[PDF Fonts]` section for PDF output, and a `[Text Fonts]` section for ASCII output. For more information about how to use fonts in reports, refer to page 8-11 in *Reports*.

Entries in these sections let you:

- Specify the fonts and point sizes that appear on drop-down menus for the Font Name and Point Size properties.
- Specify the application's default font and point size.

Point Sizes

You can specify the point sizes that appear on the Point Size property's drop-down menu with an entry that has this format:

```
point_sizes = size[ size]...
```

For example:

```
point_sizes = 8 9 10 12 14 16 18 20 24 36 48 72
```

Note: Panther uses the `point_sizes` entry only for scalable fonts. For a non-scalable font, Panther gets its available sizes from the GUI and displays these on the drop-down menu.

Default Font

You can specify the application-wide font that Panther applies when you accept Default for a screen's Font Name property with an entry that has this format:

```
default_font = font-spec
```

`font-spec` is a font specification that is valid for this configuration map file's environment.

For applications running on Windows, specify the font name only. For example:

```
default_font = Arial
```

For Motif applications, specify fonts with the `XLFD` font naming convention; substitute the wildcard character * for all weight, slant, and size properties. For example:

```
default_font = -*-Helvetica-*
```

For the PDF driver, you must specify the name of the TrueType or Type 1 font file. The file name's extension will be used to determine the type of font. The extension must be .ttf for TrueType fonts and either .pfa or .pfb for Type 1 fonts. For Type1 fonts, there must also be the corresponding .afm file in the same directory. For example, in Windows,

```
default_font = c:\windows\Fonts\times.ttf
```

and in Linux or UNIX,

```
default_font = /usr/share/fonts/default/Type1/n021003l.pfb
```

Note: At runtime, Panther uses the default font for any font specification that it cannot resolve.

Default Font Size

The `default_point_size` entry specifies the application-wide font size that Panther applies when you accept Default for a screen's Font Size property. Use the format:

```
default_point_size = size
```

Panther Font Aliases

Panther font aliases are especially useful in an application's portability across platforms. Each platform has its own configuration map, and Panther font aliases map to local font specifications. Panther font aliases appear on the Font Name property's drop-down menu. In GUI environments, Panther resolves these with the names of fonts supplied by the GUI itself.

Each Panther font alias is defined with the following format:

```
aliasName [(fontQualifier...) ] = fontSpec
          [[ (fontQualifier...) ] = fontSpec]...
```

You can specify different qualifiers for the same font alias on separate lines, and thereby map it to unique font specifications. For example, the following definition uses different qualifiers to map the Prolifics font Helvetica to two different fonts, depending on whether the Italic property is set:

```
Prolifics Helvetica (noitalic) = Arial
                    (italic)   = ArialItalic
```

If more than one entry matches a widget's properties, the first matching entry determines which font is displayed.

The following sections discuss each component of a font definition.

`aliasName`

The name that you choose to identify a font alias. Panther font aliases are defined in the configuration map file. They appear before the GUI-specific font specifiers in the Font Name property option menu.

`fontQualifier`

Optionally limits usage of `aliasName` to those objects that also use the specified qualifiers. You can `AND` together space-delimited font qualifiers from each of the following columns, in any order:

`bold`	`italic`	`underline`	*`pointSize`*[*`pointSize`*]...
`nobold`	`noitalic`	`nounderline`	

For example, a Windows configuration map file might contain two definitions for the Helvetica font, the first qualified, the second unqualified:

```
Prolifics Helvetica (italic 12 14) = ArialItalic
                                   = Arial
```

If a widget's Font Name property is set to Prolifics Helvetica, Panther uses Arial unless two other conditions are also true: the Italic property is set to Yes, and the Point Size property is set to either 12 or 14. In this case, Panther uses ArialItalic.

Point size qualifiers can limit the number of choices available in the Point Size property's option menu. For example, given the following Windows font definition, choosing Prolifics Times Roman as a widget's Font Name property limits the choices on the Point Size drop-down menu to Default, 8, and 10:

```
Prolifics Times Roman (8 10) = Times New Roman
```

Note: Point size qualifiers are used on the Point Size property's option menu only if they are valid for the selected font.

`fontSpec`

`fontSpec` maps the font name to a font supported by the GUI environment. For applications running on Windows, specify fonts with this syntax:

`fontname``[-`*`point-size`*`] [-bold] [-italic] [-underline]`

For example:

```
Prolifics Helvetica    = Arial-14-bold
Data Entry Text  = Arial
```

For Motif applications, specify fonts with the XLFD font naming convention:

```
-foundry-family-weight-slant-width-style-pixel size
-point size-x resolution-y resolution-spacing
-average width-charset registry-charset encoding
```

You can substitute any component in an XLFD font name with the wildcard character `*`. For example:

```
Prolifics Courier          = -*-courier-*-r-*
Prolifics Courier (italic) = -*-courier-*-o-*
```

If `fontSpec` omits values for point size, slant, or weight, Panther supplies these values from the corresponding property settings—Point Size, Italic, and Bold. For example, the following entries for the Prolifics Helvetica font—each in separate configuration map files for Windows and Motif—specify only the font's family name:

```
Prolifics Helvetica  = Arial
Prolifics Helvetica = -*-helvetica-*
Prolifics Helvetica = helvetica
```

Given these definitions, any widget using Prolifics Helvetica as its font can also have its Point Size, Bold, and Italic properties set; these properties are used to resolve the displayed font. So, if the widget's Bold and Italic properties are set to Yes, Panther resolves the Prolifics Helvetica to Arial-bold-italic on Windows and -*-helvetica-bold-i-* in Motif, and passes on these specifications to their respective GUIs.

Conversely, these definitions of a font named HelvBold sets its weight to bold:

```
HelvBold = Arial-bold
```

```
HelvBold = -*-helvetica-bold-*
```

The explicit weight specifications for HelvBold override the Bold properties for a widget that uses this font; the font is always displayed as bold.

For the PDF driver, you must specify the name of the TrueType or Type 1 font file. The file name's extension will be used to determine the type of font. The extension must be .ttf for TrueType fonts and either .pfa or .pfb fir Type 1 fonts. For Type1 fonts, there must also be the corresponding .afm file in the same directory. For example, in Windows,

```
Prolifics Courier = c:\windows\Fonts\cour.ttf
```

and in Linux or UNIX,

```
Prolifics Courier = /usr/share/fonts/default/Type1/n022003l.pfb
```

Sample Configuration Map File

The following configuration map file defines colors, fonts, and line styles for Windows applications.

```
[Colors]

grape           = MAGENTA       # Prolifics color
Aquatic Blue    = "64/32/200"   # Windows-style RGB value

# The following entries in the color map are for use in the
# screen editor. If you remove them entirely, then SCHEME
# colors are used, which may be desirable in Windows.

#seFormBG       = GUI WindowBackground
#seBorderFG     = Unused by Pi for Windows
#seLabelFG      = GUI WindowText
#sePushFG       = GUI ButtonText
#sePushBG       = GUI ButtonFace
#seEntryFG      = GUI WindowText
#seMultiFG      = GUI WindowText
#seMultiBG      = GUI WindowBackground
#sePwListFG     = GUI WindowText
#sePwListBG     = GUI WindowBackground
#seListFG       = GUI WindowText
#seListBG       = GUI WindowBackground
#seCheckFG      = GUI WindowText
#seOptionmenuBG = GUI WindowBackground
#seComboboxBG   = GUI WindowBackground
#seTextBG       = GUI WindowBackground

# The following definitions are for the tool box

seTbFormBG      = BLACK
#seTbBorderFG   = Unused by Pi for Windows
#seTbTogFG      = Unused by Pi for Windows

[Schemes]

#OUTPUTFG       = GUI WindowText
#TEXTFG         = GUI WindowText
#MULTITEXTFG    = GUI WindowText
#PUSHBUTTONFG   = GUI ButtonText
#TOGGLEBUTTONFG = GUI ButtonText
#RADIOBUTTONFG  = GUI WindowText
#OPTIONMENUFG   = GUI WindowText
#COMBOBOXFG     = GUI WindowText
#LISTBOXFG      = GUI WindowText
#SCALEFG        = GUI WindowText
```

```
#LABELFG         = GUI WindowText
#BOXTOPFG        = GUI WindowText
#LINEFG          = GUI WindowFrame
#CHECKBOXFG      = GUI WindowText
#FORMBORDERFG    = Unused by Pi for Windows
GRAPHFG          = BLACK
#GRIDFG          = GUI WindowText
#FORMBG          = GUI WindowBackground
OUTPUTBG         = CONTAINER
#TEXTBG          = GUI WindowBackground
#MULTITEXTBG     = GUI WindowBackground
#PUSHBUTTONBG    = GUI ButtonFace
#TOGGLEBUTTONBG = GUI ButtonFace
RADIOBUTTONBG    = CONTAINER
#OPTIONMENUBG    = GUI WindowBackground
#COMBOBOXBG      = GUI WindowBackground
#LISTBOXBG       = GUI WindowBackground
SCALEBG          = CONTAINER
LABELBG          = CONTAINER
#BOXTOPBG        = CONTAINER
#LINEBG          = Unused by Pi for Windows
CHECKBOXBG       = CONTAINER
#FORMBORDERBG    = Unused by Pi for Windows
#GRAPHBG         = CONTAINER
#GRIDBG          = CONTAINER
ACTIVEXBG        = CONTAINER
#CARDBG          = Unused by Pi for Windows
#DECKBG          = Unused by Pi for Windows

[Lines]

Style 1 = Single
MyFavoriteStyle = Double

[Windows Fonts]

# Point Size property drop-down

point_sizes = 8 9 10 12 13 14 16 18 20 22 24 26 28 36 48 72

# Application defaults for Font Name and Point Size
# properties

default_font       (print)   = Times New Roman
default_point_size (print)   = 10

# Font Name              Qualifiers   Windows font
# --------------------   ----------   ------------
Prolifics Courier                   = Courier New
Prolifics Times Roman               = Times New Roman
Prolifics Helvetica                = Arial
Prolifics Symbol                   = Symbol


[PostScript Fonts]

# Rules in this section apply only to ReportWriter's editor and
# printed output.

# Point Size property drop-down

point_sizes = 8 9 10 11 12 13 14 16 18 20 22 24 26 28 36 48 72

# Application defaults for Font Name and Point Size properties

default_font                    = Times-Roman
default_point_size              = 10


# Font Name           Qualifiers      PostScript font
# ---------           ----------      ---------------
Prolifics Courier     (italic bold) = Courier-BoldOblique
                      (italic)      = Courier-Oblique
                      (bold)        = Courier-Bold
                                    = Courier

Prolifics Times Roman (italic bold) = Times-BoldItalic
                      (italic)      = Times-Italic
                      (bold)        = Times-Bold
                                    = Times-Roman

Prolifics Helvetica   (italic bold) = Helvetica-BoldOblique
                      (italic)      = Helvetica-Oblique
                      (bold)        = Helvetica-Bold
                                    = Helvetica

Prolifics Symbol                    = Symbol


[Text Fonts]

# Rules in this section apply only to ReportWriter's editor and
# printed output.


# Point Size property drop-down

point_sizes = 8 10 12 16 24 36
```

```
# Application defaults for Font Name and Point Size properties

default_font                 = Times
default_point_size           = 10


# Font Name           Qualifiers       Text Font
# ---------           ----------       ---------
Prolifics Courier     (10 italic)      = Courier_i_10
Prolifics Courier     (10 bold)        = Courier_b_10
Prolifics Courier     (10)             = Courier_10
Prolifics Courier     (italic)         = Courier_i
Prolifics Courier     (bold)           = Courier_b
Prolifics Courier                      = Courier

Prolifics Times Roman (24 italic bold) = Times_i_b_24
Prolifics Times Roman (24 italic)      = Times_i_24
Prolifics Times Roman (24 bold)        = Times_b_24
Prolifics Times Roman (24)             = Times_24
Prolifics Times Roman (18 italic bold) = Times_i_b_18
Prolifics Times Roman (18 italic)      = Times_i_18
Prolifics Times Roman (18 bold)        = Times_b_18
Prolifics Times Roman (18)             = Times_18
Prolifics Times Roman (10 italic bold) = Times_i_b_10
Prolifics Times Roman (10 italic)      = Times_i_10
Prolifics Times Roman (10 bold)        = Times_b_10
Prolifics Times Roman (10)             = Times_10
Prolifics Times Roman (italic bold)    = Times_i_b
Prolifics Times Roman (italic)         = Times_i
Prolifics Times Roman (bold)           = Times_b
Prolifics Times Roman                  = Times

Prolifics Helvetica                    = Helvetica

Prolifics Symbol                       = Symbol
```

Translating Applications

Panther provides the following capabilities for modifying application for international usage:

- Panther uses 8-bit character data without appropriating a bit for internal use.
- The library functions `sm_dblval` and `sm_dtofield`, which read and write real values, respectively, use the C standard library functions `atof` and `sprintf` to interpret the system decimal symbol (radix character) correctly.
- The library function `sm_is_yes` uses the characters designated in the `SM_YES` and `SM_NO` entries in the Panther message file. Therefore, if you translate the message file, the screen use and display of those values are automatically internationalized. The function uses `toupper` to recognize upper-case variations.

You can also use the Panther message file to set date and time formats (page 45-17) and currency formats (page 45-19) to conform to local usage.

8-Bit Character Data

Panther supports 8-bit character data. Video files specific to the terminal can give special instructions, if necessary, on how to display international characters. This is needed if the terminal requires shifting to a different character set to display non-ASCII characters. Most terminals used in the international market do not need to shift character sets.

The video file can also be used to translate between two different standards for international characters. Thus, screens can be created with one standard and displayed using a different one.

The use of 8-bit characters for international symbols does not necessarily preclude use of graphics in character-mode applications. Unused entries in a character set, such as 0x01 through `0x1f` or `0x80` through `0x9f`, can be mapped to line graphics symbols.

Unless a widget has a keystroke filter, Panther ignores the characters that are entered into it from the keyboard. Internally, it only manipulates numbers. Cursor control keys such as arrows and `TAB`, and function keys are all assigned logical values that are outside the range `0x00` to `0xff`, and thus cannot conflict with international characters.

Keyboards that support international character sets usually produce a single 8-bit byte (perhaps with the high bit set) for each character. However, some terminals generate a sequence to represent an international character. If so, you can use a text editor to map the byte sequences to a logical value, just as the video file is used to map the logical value to the sequence required by the display terminal.

For more information on how to display non-English characters or to receive them from the keyboard, refer to page B-1 in the *Upgrade Guide*.

Translating Screens in Application Programs

There are a number of approaches to translating your application screens. If your application requires translation for international distribution, consider the following questions:

- How many translations are needed?
- Do users need access to multiple languages at runtime? When they start the application only, or during a session?
- Is the application relatively complete and static, or are changes and enhancements still be made?

The answers to these questions determine which method to use. In any event you must provide the translator with the information that needs to be translated, and pictures of the screens to provide some context. In addition, screen size and spacing should be considered when translating screens to other languages.

There are essentially three different approaches you can take to provide an application to a multilingual audience. Each approach requires some up-front planning, and some development strategy. The localization process can be performed at:

- Distribution time
- Installation time
- Runtime, which can be either at startup or dynamically at the user's request.

There are probably several ways to approach the development of a product that needs to be translated and distributed in multiple languages. One of the most obvious methods is to simply translate application screens to each of the languages that you support.

Language-specific screens can be released in a variety of ways, regardless of when the localization process takes places. For example, you can create multiple libraries; each one contains a set of screens translated to a specific language. By setting the `SMFLIBS` application variable either at distribution, installation, or runtime, you or a user can access the desired language-specific library.

The following sections describe other methods to consider when you develop an application.

Distribution Translation

A distribution translation means that when the application leaves your facility, it is released with a specific set of screens. The end user receives exactly what you send.

Method One

Develop language-specific repositories. At distribution time, use the `binherit` utility to update the content of each screen by using the appropriate repository for the required language.

Method Two

At design time, define the initial text for all widgets as a variable or token, for example `%Name%`, `%Address%`, etc. When the screens are completed, use the `f2asc` utility to convert the binary screens to ASCII format. Provide your translator with the tokenized references. Then develop a translation script that will search the ASCII file and replace the token with the translated constant. The function of the translation utility would be to find and replace tokenized text, replacing `%Name%` with `Name` for English, or Nom for the French version.

Each ASCII translation can be easily maintained and updated as screens change.

After the ASCII translations are made, they can be converted back to screens in binary format with `f2asc` and distributed accordingly.

This method has these advantages and disadvantages:

Advantages	File naming conventions can be adhered to across all libraries.
	Screen dimensions and widgets can be easily adjusted and repositioned to accommodate languages and sentence structure that might require more space on a screen.
	Adding a new language only requires a new translation.
Drawbacks	Maintenance of many different languages can be time consuming.
	You must distribute more than one library to an end user who requires more than one language.
	Languages cannot be changed dynamically at runtime.

Installation Translation

In an installation translation, the application is packaged with more than one language and the desired language is installed. You can provide an installation mechanism that lets the user set `SMFLIBS` to point to and open a library of language-specific screens.

This method has the advantage of letting users decide which language to install. On the other hand, it requires disk space to accommodate storage of multiple sets of screens; and languages cannot be changed dynamically at runtime.

Runtime Translation

In a runtime translation, users can dynamically change languages at runtime. Depending on their requirements, users might only need to select a preferred language at start up, or they might need to change languages during a session.

Method One

A start up method can be implemented in the same way described for an installation translation: you provide a mechanism that lets users choose which language to display. For example, a logon screen can provide radio buttons that correspond to each supported language, so users can choose the desired language. Their choice sets `SMFLIBS` to point to and open the appropriate library of screens.

This method has the advantage of allowing multilingual organization to run the application easily; users choose their preferred language without requiring reinstallation of the software.

This method has one possible drawback, that installation requires enough disk space to accommodate all translated screens.

Method Two

Design your screens to include all translations in one screen binary. You can do this by creating dynamic labels as scrolling arrays with only one onscreen occurrence, and then synchronizing all the label arrays on the screen, you can provide an occurrence for each language you support. The user, via a programmatic call, can scroll the array to the language of choice. For example, the third occurrence might be Italian, while the fourth occurrence is Japanese. So, if the user chooses Italian, via a screen entry function the third occurrence is displayed. If Japanese is specified, the labels can be programmatically scrolled to the fourth occurrence and so on.

This method has two advantages:

- All translations exist in one place with each screen binary.
- While working in the application, a user can choose which language to display.

This method has one possible drawback, that some translations require more space than others; screens must be designed with these limitations in mind.

CHAPTER

47 Processing the Mouse Interface

This chapter shows how to evaluate and process mouse events, mouse data, and contextual information. Topics include:

- Trapping mouse events.
- Using Panther library functions to get mouse data, such as the location of the mouse click and which buttons were pressed.
- Getting and modifying the mouse pointer's state.

Trapping Mouse Events

You can intercept single and double mouse clicks on an application-wide basis through Panther's key change hook function. You can also intercept double clicking on an individual widget through its Double Click property. Both techniques are discussed in the sections that follow.

Using Key Change Functions

With Panther's key change hook function, you can intercept single and double mouse clicks throughout the program. Panther's key file (`smkeys.h`) defines these two events through the logical keys `MOUS` for single mouse clicks, and `MDBL` for double clicks. A key change function that tests for these logical keys can use Panther library functions to examine the state of the mouse cursor and mouse buttons, and perform special processing accordingly.

For example, the following code shows in skeletal format a key change function that tests for a single click mouse event outside a field, and then determines which button, if any, is down. It also conditionally tests for different combinations of mouse events with keyboard modifiers, such as Shift+click versus Ctrl+click. Most of the processing relies on `sm_ms_inquire` to test the mouse's state. For detailed information on using this function, refer to page 47-4. For more information on key change functions, refer to page 44-36.

```
int keychg ( int which_key )
{
   int ms_btn;
   switch ( which_key )
   {
   case MOUS:

      /*is mouse click outside field? */
      if ( sm_ms_inquire( MOUSE_FIELD ) < 0
      {
         ms_btn = ( sm_ms_inquire( MOUSE_BUTTONS ) & 0x49;

         /*is any button down?*/
         if ( ms_btn > 0 )
         {
            /*test which button is down*/
```

```
                switch ( mouse_button )
                {
                   ...
                }
                /*is any keyboard modifier also down */
                if ( sm_ms_inquire( MOUSE_SHIFT ))
                {
                   ms_kbd = sm_ms_inquire( MOUSE_SHIFT );
                   switch ( ms_kbd ) /*which key is down? */
                   {
                      ...
                   }
     ...
    }
```

Trapping Double Clicks on a Widget

Several widget types have the `double_click` property, which lets you specify an action that is triggered by double clicking on a widget. `double_click` gets a control string as its value. This control string can specify to call a function, invoke an operating system command, or open another screen.

The following widget types have the `double_click` property:

- Single line text
- Dynamic label
- Combo box
- List box that is a select-any type or is defined as a selection group
- Multiline text

Getting Mouse Data

Panther provides library functions and application properties that get information about the mouse's current state:

- The mouse click's location
- The state of the mouse buttons
- Other keys that were pressed when the mouse click occurred
- The amount of elapsed time between mouse clicks

Determining Mouse Click Location

The library function `sm_ms_inquire` lets you test the last mouse click's line and column location on a Panther screen or on the physical display. Several runtime properties also offer access to the field and screen on which the last mouse click occurred.

Identifying Mouse Coordinates

To determine the line and column location of the last mouse click, supply `sm_ms_inquire` with arguments of `MOUSE_LINE` and `MOUSE_COLM`, respectively. To get the `mouse click's line and column within a Panther screen, supply` `MOUSE_FORM_LINE` and `MOUSE_FORM_COLM`. For example, the following routine gets the mouse click coordinates on a map that is displayed on a static label:

```
void get_mouse_coords( void )

{

   int longitude, latitude;


   /* make sure the user clicked somewhere on the map */
```

```
    if ( sm_ms_inquire( MOUSE_FIELD ) > 0 &&
         sm_prop_get_str
         ( PR_APPLICATION, PR_MOUSE_FIELD_NAME ) == "mapLbl" )
    {

       longitude = sm_ms_inquire( MOUSE_FORM_COLM );
       latitude = sm_ms_inquire( MOUSE_FORM_LINE );
       return get_map_location( longitude, latitude );
    }
}
```

Mouse and Widgets

The previous example also uses `sm_ms_inquire` and `sm_prop_get_str` to test whether a mouse click occurred inside a field and the field's identity:

```
if ( sm_ms_inquire( MOUSE_FIELD ) > 0 &&
     sm_prop_get_str
     ( PR_APPLICATION, PR_MOUSE_FIELD_NAME ) == "mapLbl" )
```

When supplied an argument of `MOUSE_FIELD`, `sm_ms_inquire` returns either the field number in which the mouse click occurred, or -1 if the mouse click occurred outside the field.

You can also use these runtime properties to get the name of the field and occurrence in which a mouse click occurred:

- `mouse_field` and `mouse_field_name` respectively get the number and name of the field in which the last mouse click occurred.
- `mouse_field_occ` gets the number of the occurrence in which the last mouse click occurred.

All mouse properties are application-level properties, accessible through the `@app()` modifier. For example, this all-purpose code obtains the data in the last clicked-on occurrence of any field:

```
vars data

data = @widget(@app()->mouse_field_name)[@app()->mouse_field_occ]
```

Mouse and Screen

`mouse_form_name` is an application runtime property that gets the name of the screen on which a mouse click occurred. Like other mouse properties, it is accessible through the `@app()` modifier as in this example:

```
vars mouse_screen

mouse_screen = @app()->mouse_form_name
```

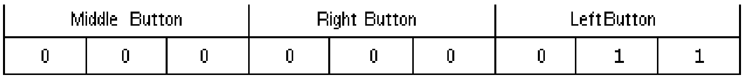
Determining Mouse Button State

You can get the state of each mouse button—up, down, just pressed, or just released—by supplying `sm_ms_inquire` an argument of `MOUSE_BUTTONS`. If successful, the function returns an integer bit mask. The function puts the requested data in three segments of three bits each, where each segment represents one of three mouse buttons—left, middle, and right. The three lowest-order bits contain left button data; if the mouse has only one button, only these bit settings are significant. The middle three bits contain right button data. 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:

- 0/1Up/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 representing 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

For example, the following routine tests whether any mouse buttons are down: it bitwise `AND's` `sm_ms_inquire's` return value with 0x49, thereby masking off all but the first, fourth, and seventh-order bits:

```
/*find out whether any button is down */
int is_any_button_down( void )
{
   return sm_ms_inquire ( MOUSE_BUTTONS ) & 0x49;
}
```

Identifying Keyboard Modifiers

By supplying `sm_ms_inquire` with an argument of `MOUSE_SHIFT`, you can find out whether a mouse click occurred with one or more of these keys pressed down: Shift, Ctrl, and Alt. The function returns an integer bit mask whose three lowest-order bits are set to indicate which of the three keys, if any, were pressed. These bits are set as follows, from lowest- to highest-order bit:

- Shift key is down
- Ctrl key is down
- 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

In the following example, the return value of `sm_ms_inquire`(`MOUSE_FIELD`) is bitwise AND's with 0x06 in order to mask off the lowest-order bit (Shift). This lets the program determine whether Alt or Ctrl, or both, were pressed down during the last mouse click:

```
if ( sm_ms_inquire( MOUSE_SHIFT ))
{
   /*test for Alt and Ctrl keys only */
   ms_kbd = sm_ms_inquire( MOUSE_SHIFT ) & 0x06;
   switch ( ms_kbd )
   {
      case 0x02: /*Ctrl key is down */
       ...
      break;
      case 0x04: /*Alt key is down*/
       ...
      break;
      case 0x06: /*Alt+Ctrl keys are down */
       ...
      break;
   }
```

Reporting Elapsed Time between Mouse Clicks

`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.

Notes: Ordinarily, you can use the key change function to intercept double mouse click events. For more information, refer to page 47-1.

Changing the Mouse Pointer State

`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 supplied argument. It can also specify to change the cursor's state automatically, depending on whether the application is awaiting input or not.

For GUI platforms, you can set a screen's default cursor through its 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_delay_cursor` takes a single integer argument, one of these constants:

`SM_AUTO_BUSY_CURSOR`
: Sets the mouse pointer to toggle automatically 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.

CHAPTER

46 Dynamic Data Exchange

Panther supports Windows Dynamic Data Exchange (DDE), which lets applications share data through client/server links. Panther supports both client and server links with other applications. As a DDE server, a Panther application exports data from a field. As a DDE client, it imports data from another application into a field.

Panther as a DDE Server

As a DDE server, Panther can export data from a named field to a client application. The client application specifies a DDE service, topic and item. In Panther, these correspond to the application name, screen name and field name, respectively.

For example, an Excel spreadsheet can request a link between one of its cells and a Panther field. The request must include the name of the Panther server name, the screen name, and the field name. If the request succeeds, a DDE connection is created between that spreadsheet window and the Panther screen; this connection initially consists of the requested link, and can also accommodate other links that the client might request later.

Enabling Connections

Before a client application can request links to a Panther application, two conditions must be true:

- The Panther application must be running.
- The application must be enabled as a DDE server.

To enable Panther as a server, call `sm_dde_server_on` or include this setting in the initialization file `prol5w32.ini` or `prol5w64.ini`:

```
DDEServer=on
```

Creating Links

Clients can create links to a Panther screen either through the clipboard or by explicitly issuing a request.

Paste Links

You can use the Windows clipboard data to create a link to a Panther application:

1. Copy data from a Panther field to the clipboard. The clipboard data includes the link information required by DDE: service, topic, and item.
2. Paste link the clipboard data into the client application. The link information that is embedded in the pasted data initiates a link request from this client to the originating Panther screen.

Links Specified in Client Syntax

You can also create links to a Panther screen through explicit requests in the client application's DDE syntax. Refer to your client application's documentation for details on its DDE syntax.

Notes: Prefix the topic name with ampersands (& or &&) if you want the Panther screen to open as a stacked or sibling window.

The following examples show DDE syntax for two applications:

Microsoft Word

```
{DDEAUTO myProlApp &mainScreen totalData }
```

Microsoft Excel

```
=myProlApp|mainScreen!totalData
```

Processing Links

It is the client application's responsibility to connect with a Panther screen and create links to the required fields. Most client applications automatically establish a connection when they request a link.

When DDE gets a link request that is intended for a Panther application, it processes it as follows:

1. Finds a match between the specified service and a Panther application that is already running.
2. Checks whether the Panther application is enabled as a DDE server.
3. Matches the specified topic to a screen and checks whether a connection already exists.
4. Opens the screen, if necessary, and matches the specified item to a field.
5. Creates a link between the client and the field.

Links remain in effect until they are explicitly closed by the client or the Panther application exits. Panther maintains links for a screen that is inactive or closed, and resumes data updates when the screen reopens. The client's connection to a Panther screen remains active until all links to the current screen are destroyed.

Panther destroys links only at the request of the client or when the Panther application terminates. When the client destroys its last link to a screen, its connection to that screen is broken.

Updating Client Data

A client can create three kinds of links to Panther fields:

Hot Links

Panther automatically updates the client with new data as soon as linked field data changes. Hot links are maintained only for fields on the active screen.

Warm Links

Panther notifies the client when linked field data changes. The client must then request the data. Notice is sent only for fields on the active screen. Requests for data succeed only if the linked field is on an active screen or in the LDB.

Cold Links

Panther updates the client with new field data only when the client requests it. Requests for data succeed only if the linked field is on an active screen or in the LDB.

Array Data

If the linked field is an array, Panther supplies all occurrences in the array. Occurrences are separated by carriage returns () and newlines (). Leading blanks in right-justified fields and trailing blanks in left-justified fields are omitted.

Data Conversion

Panther supplies text data to client applications. It is the client's responsibility to perform any necessary data conversion such as string to numeric. For example, if the client is a Microsoft Excel spreadsheet, the spreadsheet cell should wrap the formula for the DDE reference in a value function call as below; this converts the link text data into a number:

```
=value(JAM|screen.jam!textdata)
```

Other methods are specific to each client application.

Disabling Panther as a DDE Server

You can disable Panther as a server at runtime through `sm_dde_server_off`. Panther continues to maintain all previous links to clients; however, it ignores all client requests that are made after this call.

Panther as a DDE Client

When a Panther application acts as a DDE client, it imports data into fields from an external server application. Panther can request hot, warm, and cold links.

DDE can maintain multiple connections between the Panther application and server application; only one connection is allowed between a Panther screen and a given server window. Each connection can maintain multiple links.

For example, a Panther screen can request a hot link between one of its fields and a cell in an Excel spreadsheet. The request must include the name of the Excel program, the spreadsheet's filename, and the cell identifier. If the request succeeds, a DDE connection is created between the Panther screen and the specified spreadsheet; this connection initially consists of the requested link, and can also accommodate other links that the client might request later. Subsequent changes in the server cell data are reported to Panther and automatically are written to its linked field.

Enabling Connections

Before Panther can connect to a server application, it must be enabled as a DDE client. To enable Panther as a client, call `sm_dde_client_on` or include this setting in the initialization file `prol5w32.ini` or `prol5w64.ini`:

```
DDEClient=on
```

Creating Links

As a DDE client, Panther can request hot, warm, and cold links. For more on link types, refer to page 46-3. You can create links in three ways:

- Call one of Panther's paste link functions, which get server data from the clipboard.
- Specify server data with one of Panther's connect functions.
- Specify server data in the initialization file.

Paste Links

You can paste server data from the Windows clipboard into a Panther field on the active screen with one of these Panther paste link functions:

- `sm_dde_client_paste_link_hot`
- `sm_dde_client_paste_link_warm`
- `sm_dde_client_paste_link_cold`

These functions take a single argument—the field to get server data. Panther gets the actual data and link information from the clipboard—server, topic, and item—and paste links the data into the specified field.

Explicit Links Through Library Functions

Panther also provides a set of library functions that explicitly specify the server data required:

- `sm_dde_client_connect_hot`
- `sm_dde_client_connect_warm`
- `sm_dde_client_connect_cold`

These functions take four arguments: the server, topic, item, and target Panther field. The format for server, topic, and item arguments is specific to each server application. Refer to the server application's documentation for this information. on page 46-7 shows the syntax used by three widely used Windows applications.

Table 46-1 Sample server syntax for Windows applications

	Quattro Pro	MS Word for Windows	MS Excel
Server	QPW	Winword	Excel
Topic	`C:\SALES.WB1`	`C:\WORK\SALES.DOC`	`C:\XL\SALES.XLS`
Item	`$A:$B$1..$B$1`	`DDE_LINK1`	`R1C2`

For example, the following JPL statement creates a cold link between a Panther client and an Excel spreadsheet:

```
retval = sm_dde_client_connect_cold \
>          ("Excel","C:\XL\SALES.XLS","R1C2","total")
```

Links Specified in Initialization File

You can use the initialization file to create hot links. This lets you specify and edit links for an application without changing the screens themselves. Only hot links are supported from the initialization file.

The initialization file for Panther contains a Panther DDE section, where you can specify links to server applications as follows:

```
screenname !

 fieldname =

 service |

 topic !

 item
```

For example, a link to a Quattro Pro spreadsheet might look like this:

```
salesScrn!totalSales=QPW|C:\MyAcct\Sales.wb1!$A:$A$10..$A$10
```

The format for server, topic, and item arguments is specific to each server application. Refer to the server application's documentation for this information. Table 46-1 shows the syntax used by three widely used Windows applications.

Processing Link Requests

A link request from a Panther application consists of these steps:

1. DDE checks whether the server application is running and the specified topic is open. Both conditions must be true; otherwise, the link request fails.
2. DDE checks whether a connection already exists between the server topic and the requesting Panther screen. If none exists, DDE attempts to create one.
3. After DDE verifies or establishes a connection, it creates the specified link—hot, warm, or cold—between the specified Panther field and the server item. It then updates the field data according to the link type.

If a link request fails for any reason—for example, because the server application is not running—Panther posts an error message.

Updating Data from the Server

Panther updates link data according to the link type:

- Hot link—Data is updated on the client whenever it changes on the server.
- Warm link—The server notifies the client of a change in data, but sends new data only at the client's request.
- Cold link—Data is updated only at the client's request. The server does not notify the client of data changes.

If a field has warm or cold link data, the application must explicitly request updates from the server by calling `sm_dde_client_request`. This function can be called only for fields on the active screen.

When warm link data changes, DDE notifies Panther that new data is available from the server. Panther then calls a callback function—either its own or one installed by the developer—and passes it the screen name and field name of the link. For information about writing callback routines, refer to `sm_dde_install_notify`.

DDE does not notify the Panther client of any changes in cold link data.

Notes: Because `sm_dde_client_request` can be called only for widgets on the active screen, an application that uses warm links should queue notices for data on inactive screens.

Array Data

Panther tries to update all occurrences in the array with server data. Data flows into the array starting with the first occurrence. When Panther reaches the end of the occurrence or encounters a tab, carriage return, or newline in the server data, it skips to the next occurrence. Panther eliminates leading white space—tabs, carriage returns, and new lines—before writing the data. The process ends when there is no more data or the end of the array is reached.

Destroying Links to a DDE Server

When a screen closes, Panther destroys all links on that screen. You can also explicitly destroy links on the active screen with `sm_dde_client_disconnect`.

Disconnecting from a DDE Server

Panther maintains its connection to a server application as long as an open screen contains a link to that application. When the last screen containing a link to a server closes, Panther breaks the connection.

Execute Transactions

The execute transaction lets a client execute a command on a server. As a client, Panther can initiate execute transactions on a server application; as a server, Panther can be the recipient of commands issued by a client.

As a DDE client, Panther can execute a command on a server with which it already has a connection by calling `sm_dde_execute`:

```
sm_dde_execute(server,topic,command);
```

The server decides whether to execute or ignore the command. You can check the function's return value to determine the outcome of the call.

As a server, Panther can receive a command issued by a client. For example, a Quattro Pro spreadsheet might contain this `EXECUTE` statement:

```
{EXECUTE B1, "^updateData.jpl"}
```

Panther executes the command string like any control string.

For information about specifying execute transactions from client applications, refer to that application's documentation.

Poke Transactions

A poke transaction lets a client send data to a server. As a client, Panther can initiate poke transactions on a server application; as a server, Panther can be the recipient of commands issued by a client.

As a DDE client, Panther can poke data into a server with which it already has a connection by calling `sm_dde_poke`:

```
sm_dde_poke(server,topic,item,data);
```

The server decides whether to execute or ignore the command. You can check the function's return value to determine the outcome of the call.

As a server, Panther can be the target of a poke transaction issued by a client. For information about executing poke transactions from client applications, refer to that application's documentation.

CHAPTER

48 Writing Portable Applications

This chapter describes features of hardware and operating system software that can cause Panther to behave in a non-uniform fashion. If you want to create and write programs that run across a variety of systems, you need to be aware of these factors.

Panther Header Definitions

The header files smmach.h and smcommon.h contain information that library functions need in order to deal with certain machine, operating system, and compiler dependencies. These include:

- The presence of certain C header files and library functions.
- Byte ordering in integers and support for the unsigned character type.
- Path name and command line argument separator characters.
- Pointer alignment and structure padding.

The header files are thoroughly commented. Follow the directions in the file and use the information that applies to your machine and operating system.

Terminal Dependencies

Some general differences among terminals are described in this section. Recommendations and considerations are included to help ensure that your application can be ported to terminals that differ from your development environment.

Display Area and Attributes

Panther can run on display terminals of any size. On character-based terminals without a separately addressable status line, Panther uses the bottom line of the display—typically, line 24—for a status line, and status messages overlay that line's contents. To ensure enough room for status line and screen displays, design for an average screen size of 23 lines by 80 columns, including the border.

Different terminals support different sets of attributes. Panther makes sensible compromises based on the attributes available. However, do not design programs that rely extensively on attribute manipulation to highlight data, which might not be evident on terminals with an insufficient number of attributes. For example, colors do not display on monochrome terminals. On the other hand, consider designating the appropriate color combinations in the event that your application is ported to terminals that support color. Also, use of graphics character sets is especially terminal-dependent.

Attribute handling can also affect the spacing of fields and text. In particular, if you design screens to run on terminals with onscreen attributes, leave space between fields, highlighted text, and reverse video borders for the attributes. Some terminals with area attributes also limit the number of attribute changes permitted per line or per screen.

Use color aliases to ensure cross-GUI color compatibility.

Key Translation and Labels

The key translation table mechanism supports the assignment of any key or key sequence to a particular logical character. However, if you make excessive use of function keys for program control, the number and labelling of function keys on particular keyboards can cause constraint. For instance, the standard VT100 has only four function keys. In this case, consider using menus rather than function keys to implement choice among alternatives.

Use key labels in your key translation file instead of hard-coded key names. This helps ensure portability to a variety of terminals. With the %K escape, the key labels can be automatically inserted in field status text and other status line messages. To include the key name in a field, use `sm_keylabel` to return a printable name of the logical key.

Language Dependencies

To a large extent, Panther depends on the operating system to handle discrepancies between different languages and their character sets. The following sections discuss some of the system-related factors that you might need to account for when preparing an application for different languages.

You can also support different languages by modifying Panther' message file. (See page 45-2, “Using Message Files.”) For strategies to translate screens into different languages, refer to page 45-46, “Translating Screens in Application Programs.”

Keystroke Filter Translation

Panther evaluates user input only when a widget's `keystroke_filter` property is set to any value other than `PV_UNFILTERED`: A keystroke filter restricts user input—for example, to digits only, yes/no, or to conform with an edit mask or regular expression. To validate data as it is entered, Panther uses standard C macros, such as `isdigit` and isalpha. Panther relies on the operating system to supply these macros in a form suitable for international use. Absent operating system support, care should be taken when using these capabilities.

Panther uses its own code to process numeric entry because C does not provide an macro to evaluate real numbers. If the widget has its decimal_symbol property set, Panther uses it to evaluate numeric input.; otherwise, Panther uses the decimal symbol that is set in the message file or derived from the operating system.

Widgets that are restricted to yes/no entry use the characters that are specified by the message file's SM_YES and SM_YES entries. (See page 45-24, “Setting Yes/No Values.”) Although some vendors supply information about these characters, the ANSI standard leaves the issue unresolved. Therefore, Panther relies on the message file to evaluate this data.

If the keystroke filter specifies a regular expression (keystroke_filter = PV_REGULAR_EXP), Panther uses the ASCII collating sequence to validate ranges of characters. Therefore, this expression matches only English lower-case letters:

```
[a-z]*
```

The European character ä, for example, is not matched by this expression.

Case Conversion

Widgets that have their convert_case property set to PV_UPPER or PV_LOWER depend on the toupper and tolower functions. The present code assumes that the return from toupper is appropriate for conversion to upper case. A lower case letter that has no upper case equivalent—for example, the German “double s”—remains in lower case.

Range Checks

Numeric Data

Range checks for numeric data are handled by the C library routine at of (assuming that the “strip” routine works properly).

Alphabetic Data

Collating sequences can vary among different languages. This is especially true for dictionary or telephone book processing. For example, upper- and lower-case letters are compared equally. Also, a telephone book evaluates St. and Saint as equal and ignores hyphens. Some languages can pose problems even for less demanding applications. For example, ligatures compare equally to pairs of letters. The placement of vowels with diacritical marks varies widely even among countries using the same language.

The ANSI standard specifies the routine `strcoll` to expand a word into a format that `strcmp` can use for comparisons. These routines assume that the data supplied is a word in the local language. They can yield unexpected results on non-language data.

Panther is not designed to process languages in a way that involves such fine distinctions. It does sort names of fields and other objects, but only to speed look-up. As long as the sort routine and the search routine use the same algorithm, things will work.

Non-Language Data

In Panther, range checks can be applied to non-language data. For example a menu selection might have a range of a to d. In certain languages an umlaut falls into that range if a language-specific comparison is made. In cases like these, you might need to create different screens for each language.

The C routines `strcmp` and `memcmp` are used to range check on non-language data. These routines compare the internal values of the characters, without regard to their actual meaning.

CHAPTER

49 Sending Mail in Panther

In Panther 5 for Windows, you can send mail. Mail is available to all Windows Panther components including COM/MTS and EJB components. Starting with Panther 5.40 you can also use CDO (Collaborative Data Objects) to send mail. This may work better than MAPI (the Windows Messaging API) on Windows Server systems.

This feature uses a combination of properties and functions. There is a global mail object `PR_MAIL_SYSTEM` that accepts the information needed to communicate with `MAPI` or `CDO`. Using the `PR_MAIL_SYSTEM` object, you can also set default values for some mail properties.

The `sm_mail_new` function creates a new mail object; other functions attach files to the message, get the message text from a field or a text file, and send the mail message.

The `sm_mail_message` function can send simple mail messages.

Defining Global Mail Properties

The global mail object `PR_MAIL_SYSTEM` is created at startup. You can set values for the following properties:

`PR_MAIL_PROFILE`
    Optional `MAPI` profile name for the `MAPILogon` function.

`PR_MAIL_SERVER`
: The mail server to use by CDO, for example "smtp.gmail.com".

`PR_MAIL_PORT`
: The port that CDO will use when connecting to the mail server. The default is to use port 25. Other commonly used ports are 465 and 587.

`PR_MAIL_USERNAME`
: The UserName CDO should use when connecting to the mail server. If omitted, anonymous SMTP will be used. PR_MAIL_PASSWORD, when set, will be used as the password for UserName.

`PR_MAIL_PASSWORD`
: Optional password for the `MAPILogon` function and for CDO when the `PR_MAIL_USERNAME` property is set.

`PR_MAIL_FLAGS`
: Optional flags that describe how to logon to the `MAPI` session or the CDO option to use. One or more can be used. Values are:

  `PV_MAPI_NEW_SESSION`
  : Always create a new session. By default, logging into an existing session will be attempted and if that fails, then a new session will be created.

  `PV_MAPI_LOGON_UI`
  : Display a logon dialog asking for the logon information when logging onto a MAPI session. If `PR_MAIL_PROFILE` is set, it will be used as the default profile name.

  `PV_MAPI_PASSWORD_UI`
  : Display a password dialog when logging onto a `MAPI` session. This setting is ignored if `PV_MAPI_LOGON_UI` is also set.

  `PV_MAPI_USE_CDO`
  : Use CDO when sending mail. The flags PV_MAPI_NEW_SESSION; PV_MAPI_LOGON_UI and PV_MAPI_PASSWORD_UI must not be set. PR_MAIL_SERVER and PR_MAIL_FROM must be set before mail can be sent.

  `PV_MAPI_USE_CDO_SSH`
  : CDO should use SSH when communicating with the mail server.

`PR_MAIL_CONNECTED`
: Indicates whether there is an active `MAPI` connection to the messaging system. It can be set to `PV_YES` to attempt such a connection and `PV_NO` to sever the current connection. Panther attempts to connect to a `MAPI` session when a

message is being sent or when the `PR_MAIL_CONNECTED` property is set to `PV_YES`. It will stay connected until any of the `PR_MAIL_PROFILE`, `PR_MAIL_PASSWORD` or `PR_MAPI_FLAGS` properties are set; until `PR_MAIL_CONNECTED` is set to `PV_NO`; or until Panther exits. It will have value `PV_NO` when the `PV_USE_CDO` flag is set.

You can set default values for all mail communication for the following properties:

`PR_MAIL_FROM`
: Default value for the `From:` line of mail messages.

`PR_MAIL_TO`
: Default value for the `To:` line of mail messages.

`PR_MAIL_CC`
: Default value for the `CC:` line of mail messages.

`PR_MAIL_BCC`
: Default value for the `BCC:` line of mail messages.

`PR_MAIL_REPLYTO`
: Default value for the Reply-to: line of mail messages. Ignored when using MAPI.

`PR_MAIL_SUBJECT`
: Default value for the `Subject:` line of mail messages.

`PR_MAIL_RECEIPT`
: Default value for whether to ask for a receipt. This seemingly does not work for most Mail Transfer Agents (Outlook and Outlook Express in particular).

Defining Multiple Addresses

`PR_MAIL_TO`, `PR_MAIL_CC` and `PR_MAIL_BCC` can include several addresses. For example:

```
@id(PR_MAIL_SYSTEM)->mail_to = \
'John Doe <j.doe@somewhere.com>, Jane Roe <j.roe@elsewhere.com>'
```

Such items will be split up internally, so

```
   vars first_to = @id(PR_MAIL_SYSTEM)->mail_to[1]
```

would return the value `'John Doe <j.doe@somewhere.com>'` after the above.

Creating and Sending Email

Creating a Mail Message Object

A new mail message object is created by calling `sm_mail_new`:

```
int sm_mail_new (char *name);
```

`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 also 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 will be ignored by some Mail Transfer Agents (Outlook and Outlook Express in particular).

PR_NAME
Name of the message.

When any of PR_MAIL_FROM, PR_MAIL_TO, PR_MAIL_CC, PR_MAIL_BCC, PR_MAIL_RECEIPT or PR_MAIL_SUBJECT are not set, the corresponding value from the PR_MAIL_SYSTEM object will be used when it is set. Setting a property to '' is treated as setting the value, i.e.,

```
@id(my_first_message)->mail_bcc = ''
```

PR_MAIL_SUBJECT, PR_MAIL_FROM, PR_MAIL_TEXT, PR_MAIL_REPLYTO, PR_MAIL_RECEIPT and PR_NAME have only one occurrence; the other properties can occur several times.

Sending Mail

sm_mail_send sends the message identified by obj_id or name and deletes it. Returns 0 if successful or an error code from smuprapi.h.

```
int sm_mail_send (int obj_id);

int sm_n_mail_send (char *name);
```

Sending a Screen Image

sm_mail_widget can only be used in prodev and prorun. The screen image of a widget is converted to a JPEG file that is attached to the mail message. PR_APPLICATION will send the complete MDI frame. Returns 0 if successful or an error code from smuprapi.h, most likely PR_E_OBJECT or PR_E_OBJID.

```
   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);
```

Sending Mail Using a Field

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.

```
int sm_mail_text (int obj_id, char *field_name);
int sm_n_mail_text (char *name, char *field_name);
```

Sending Mail from a Text File

sm_mail_file_text takes the message text from a text file.

```
int sm_mail_file_text (int obj_id, char *file_name);
```

```
int sm_n_mail_file_text (char *name, char *file_name);
```

Sending Simple Mail Messages

sm_mail_message mails a message containing text with to as the address.

```
int sm_mail_message (char *to, char *subject, char *text);
```

The default values of PR_MAIL_FROM, PR_MAIL_CC, PR_MAIL_BCC and PR_MAIL_RECEIPT will be used of they are set.

Sending Attachments

The following functions attach a file to the 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);
```

pathname is the path to the file. filename is an optional filename 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. If delete is not zero, the file will be deleted when the message is sent or deleted.

File attachments can only be created with sm_mail_attach or sm_n_mail_attach. Once created, these properties can be accessed and set.

PR_MAIL_ATTACHMENT_PATHNAME
: File path and name to be sent as the attachment.

PR_MAIL_ATTACHMENT_FILENAME
: File name to be sent with the attachment. If not set, the file name from PR_MAIL_ATTACHMENT_SOURCE will be used. Not used when sending mail using CDO

`PR_MAIL_ATTACHMENT_DELETE`
: If set to `PV_YES`, the file that is to be attached will be deleted when the mail object is destroyed.

`PR_MAIL_ATTACHMENT_ENCODING`
: Identifies the encoding used for the attached file.

`PR_MAIL_ATTACHMENT_TAG`
: Indicates the application that generated the attachment. Typical values are "text/html"; "text/plain"; "image/jpeg" and "application/pdf".

APPENDIX

A Development Utilities

This appendix describes utilities that are useful in the development process:

- `bin2c`—converts binary files into C data structures.
- `bin2hex`—converts binary screens to hexadecimal ASCII file.
- `binherit`—updates screens and reports with inherited property values from repositories.
- `cmap2bin`—converts configuration map files to binary.
- `f2asc`—converts screens, service components and reports between binary and ASCII format.
- `formlib`—creates and maintains libraries and repositories.
- `jif2asc`—converts JIF between binary and ASCII format (JetNet and Oracle Tuxedo).
- `jpl2bin`—converts JPL modules between binary and ASCII format.
- `m2asc`—converts menu files between binary and ASCII format.
- `msg2bin`—converts ASCII message files to binary format.
- `msg2hdr`—creates header files for user messages.
- `s2asc`—converts styles files between binary and ASCII format.

bin2c

Converts binary files into C data structures

```
bin2c [-fluv] asciiFile inputFile ...
```

`-f`
: Overwrite an existing output file.

`-l`
: Convert filenames sent to output to lower case.

`-u`
: Make array of unsigned chars instead of chars.

`-v`
: Generate list of files processed.

`asciiFile`
: Name of the output file.

`inputFile`
: Name of the input file.

Description

`bin2c` converts Panther binaries—such as screens, menus, and JPL modules—into C character arrays. When `bin2c` creates the ASCII C file, it generate an array for each of the input files. An array in the file has one of these two forms:

```
char inputFile[] = { contents of file };
unsigned char inputFile[] = { contents of file };
```

where `inputFile` is the name of the source binary file with its path and extension stripped off. If you use the -l option, `inputFile` is converted to lower case.

Files created with `bin2c` arrays can be compiled, linked with your application, and added to the memory-resident form list with `sm_formlist`. For more information on memory-resident lists, refer to this function and to page 42-8. The following files can be made memory-resident:

- Screens

- JPL modules
- Menus
- Key translation files
- Setup variable files
- Video files
- Message files

You cannot convert a file back to its original binary form after using `bin2c`. Panther provides other utilities that permit two-way conversions between binary and ASCII formats. For screens, these utilities are `bin2hex` and `f2asc`.

bin2hex

Converts binary screens to hexadecimal ASCII file

```
bin2hex -c[flv] asciiFile screen ...
bin2hex -x[flv] asciiFile
```

`-c`	Create an ASCII file from one or more screens.
`-f`	Overwrite an existing file.
`-l`	Convert filenames in output to lower case.
`-v`	Generate list of files processed.
`-x`	Extract all screens contained in an ASCII source; selective extraction is not supported.
`asciiFile`	Specifies the name of the ASCII output file or ASCII input file if using the -x option.
`screen`	Name of a screen to convert to hexadecimal.

Description

`bin2hex` converts binary files to and from hexadecimal to let you port Panther screens across different systems. By default, the screen editor creates binary screen files.

With the -c option, all named binary input files (`screen`) are converted to hexadecimal ASCII and added to `asciiFile`. Path names are stripped off; extensions are left intact. With the -x option, `bin2hex` extracts each screen in the specified `asciiFile` and puts each file, in binary format, in the current directory.

Selective extraction of screens from `asciiFile` is not supported. Only one argument is supported with the -x option; additional arguments are ignored.

binherit

Updates screens and reports with inherited property values from repositories

```
binherit [ -r repository ] [ -v level ] [ -u ] -l library [
    libMember ...]
```

`-l library`
: Name of the library from which to read for updating all members or individual library members (*`libMember`*).

`-r repository`
: Name of the repository from which to inherit. If not specified, `binherit` checks the value of `SMDICNAME`. If the variable is not set, the utility seeks a repository named `data.dic` in the current directory and along `SMPATH`. If unable to find the repository, an error is issued.

`-u`
: Update the screens/reports as well as listing the differences.

`-v level`
: Specify level of detail to report: `0`: No reporting, `1`: List screens as they are processed (the default setting), `2`: List screens and widgets as they are processed, `3`: List screens, widgets, and properties as they are processed.

Description

`binherit` opens the named library and searches all or specified library members for widgets having the Inherit From property set to a repository entry in the open repository, for example, `titles!title_id`. For those widgets, it compares the inherited property values with the values in the repository. The properties that have inheritance disabled are ignored.

If the screen's Inherit From property is set, it compares the values in the inherited screen properties with the corresponding values in the repository.

If the `-u` option is specified, it updates the file with the repository value; however, this option will be ignored for members of libraries that can only be opened read-only.

Errors

The following table describes possible errors, their causes, and the corrective action to take:

No repository is open.	
Cause	Repository could not be opened either because: the name specified after the `-r` option could not be found or opened, the `-r` option was not used and the value in `SMDICNAME` was not set or found, or a repository having the name `data.dic` could not be found.
Action	Use the `-r` option, ensuring that the spelling and location of the specified repository is correct.
Not a Panther repository.	
Cause	File specified after the `-r` option was incorrect.
Action	Check the spelling and location of the specified repository.
Unable to inherit property *propertyName* for *objectId*	
Cause	The object listed in the Inherit From property cannot be found in the current repository.
Action	Make sure the current repository was specified. Also, check the Inherit From property for the object.
Unable to open Panther library.	
Cause	Unable to find or open the specified library.
Action	If the library is not in the current directory, include the path name.
Unable to open Panther repository.	
Cause	Unable to find the specified repository.
Action	If the repository is not in the current directory, include the path name.
Verbosity (-v) must be 0, 1, 2, or 3	
Cause	An invalid value followed the `-v` option.
Action	Supply one of the listed values in the command line.

cmap2bin

Converts configuration map files to binary

```
cmap2bin [-pv] [-e ext] mapFile ...
```

-e *ext*
: Use the specified *ext* extension in the output file name instead of the default `bin` extension.

`-p`
: Place the output file in the same directory as the input file.

`-v`
: Generate a list of files processed.

mapFile
: Name of ASCII configuration map file; more than one input file can be specified.

Description

`cmap2bin` converts one or more ASCII configuration map files to a binary format for use by Panther.

`cmap2bin` automatically appends the binary file name with the `bin` extension unless you specify a different extension with the `-e` option. It places the binary output file in the directory where you run the utility unless you use the `-p` option.

Errors

The following table describes possible errors, their cause, and the corrective action to take:

Attribute %s not allowed in a background scheme.	
Cause	Attribute specification cannot be applied to a particular background scheme.
Action	Remove the attribute specification and rerun `cmap2bin`.

Background attributes %s in extended color definitions.

Cause	Background attributes apply only to Panther basic colors.
Action	Remove background attribute specification and rerun `cmap2bin`.

Background color %s in a foreground scheme

Cause	A background color specification (begins with `B_`) was named in a foreground scheme.
Action	Edit foreground scheme to use a foreground color specification and rerun `cmap2bin`.

CONTAINER attribute not allowed in foreground scheme

Cause	Container specification was indicated in a foreground scheme.
Action	Edit foreground scheme that uses `CONTAINER` specification and rerun cmap2b`in.`

CONTAINER is used for background only.

Cause	Container specification was indicated.
Action	Edit foreground scheme that uses `CONTAINER` specification and rerun cmap2b`in`.

Error opening file '%s'.

Cause	An error was encountered when attempting open the specified file.
Action	Confirm that the file is readable and that the target directory can be written.

Extra equal sign

Cause	A line in the configuration map file includes an extra equal sign in the specification.
Action	Correct the line and rerun `cmap2bin`.

Illegal color scheme name %s

Cause	Color specification cannot be resolved.
Action	Check *`mapFile`* for illegal color name, edit the file and rerun `cmap2bin`.

Missing equal sign

Cause	A line in the configuration map file has no equal sign following the tag.
Action	Correct the input and rerun `cmap2bin`.

No other attribute allowed if CONTAINER is specified

Cause	A display attribute was assigned to a background color scheme identified as `CONTAINER`. A `CONTAINER` specification means the object adopts the color and attributes of it container.
Action	Either remove the attribute specification or change the `CONTAINER` specification to a specific color.

f2asc

Converts screens, service components and reports between binary and ASCII format

```
f2asc -a[cfn] [-i headerFile] asciiFile binaryFile ...
f2asc -b[f] asciiFile
```

`-a`
: Create ASCII listing of one or more screens, service components and/or reports.

`-b`
: Extract all binary files (screens, service components, reports) from an ASCII listing. This option does not accept an output filename.

`-c`
: Do not generate comment lines (`-a` option only).

`-f`
: Overwrite an existing file.

-i *`headerFile`*
: Include specified *`headerFile`* at beginning of ASCII output.

`-n`
: Do not sort PI edits in the ASCII file (backwards compatibility option).

`asciiFile`
: With `-a` option, name of the file to receive ASCII version of *`binaryFile`*. With `-b` option, name ASCII file to convert back to binary format.

`binaryFile`
: Filename of screen, service component or report to convert to ASCII.

Description

`f2asc` lets you create an ASCII listing of the contents of a screen, service component or report (`-a` option); and convert it back to binary format using the `-b` option. The editor creates and uses binary files only.

With `-a`, you must specify the name of at least one screen/component/report (or use wildcard characters). With `-b`, names are ignored. The `-b` option automatically extracts all files from the specified `asciiFile.f2asc` is typically used for documenting applications. It is also useful for editing tasks that are best performed by text editors—for example, global search and replace operations.

ASCII Output

The text file generated by `f2asc` describes the contents of the screen, service component or report—the widgets that compose it and their respective properties. It is broken into sections by object type, starting with the screen/component/report itself, and identifies the object by name, if it has one. Subsequent statements in the section describe each object through attribute keywords.

`A:`*`layoutAreaName`*
: A layout area in a report.

`B:`*`widgetName`*
: A non-field widget: box, line, grid, graph, report area or tab deck.

`C:`*`logicalKey`*
: Identifies a logical key that is associated with a control string. The subsequent `ACTION` statement contains the control string itself.

`F:`*`fieldName`*
: The name of a field (including text, check box, push button, link, dynamic label, and dynamic output widgets).

`G:`*`groupName`*
: Selection group.

`I:`
: Service component's interface definition.

`L:`*`staticLabelName`*
: Static label.

`P:`*`splitterName`*
: Information about a splitter and the panes it contains.

`R:`*`reportName`*
: Name of the report. All report files begin with this entry.

`S:`*`screenName`*
: Name of the screen. All screen files begin with this entry.

`T:`*`tableViewName`*
: Table view widget.

`Y:`*`syncGroup`*
: Synchronized scrolling group.

Comment lines begin with a pound (#) character. For example:

```
#  NUMBER=1
```

Two types of expressions are used to specify the properties:

- A boolean expression is a string that sets a property to be on or off; its absence implicitly sets the same property to the opposite value. For example, `CIRCULAR` specifies that an array's circular property is set to `PV_YES`; its absence means that circular is set to `PV_NO`. The following statements contain two boolean expressions: the first sets a widget's `autotab` property to `PV_NO`; the second sets its `input_protection` property to `PV_YES`:

  ```
  NO-AUTOTAB
  PROTECTED FROM DATA-ENTRY
  ```

- An assignment expression explicitly assigns a value to a property. For example, the following statements assign values for several widget Geometry properties:

  ```
  LENGTH=15 ARRAY-SIZE=5 VERT-DISTANCE=1
     MAX-LENGTH=255 SHIFT-INCR=8
  ```

There are two types of keywords describing object properties, flags and values:

- A flag keyword is by itself and requires no other information—for example the `NUMERIC` keyword represents the numeric field type property and needs no value. A flag keyword can appear on the same line as other keywords.
- A value keyword must be accompanied by more information—it is followed by and equals sign (=) and a value represented by another keyword or a number or string. For example `GROUP=group1` shows that a field belongs to `group1` of a screen. Value keywords that begin with PI describe graphical properties of an object.

formlib

Creates and maintains libraries and repositories

```
formlib -c [-fluv] library [filename ...]
formlib -r [-luv] library filename [filename ...]
formlib -x [-fluv] library [memberName...]
formlib -{d|t} [-luv] library [memberName ...]
formlib -{e|i|m|o|s|w|z} [-v] library
formlib -g cfgStr [-v] library
```

`-c`

Create a new library/repository that optionally contains the files named (*`filename ...`*). If no files are specified, an empty library is created.

`-d`

Delete the named members from the library/repository.

`-e`

Configure the library/repository for external file locking.

`-f`

Overwrite a library member or repository entry.

`-g` *`cfgStr`*

Define a configuration management string; begins with the name of your configuration management system—either `sccs`, `pvcs` or `scpi` (in lower case). For information about source control management options, refer to page 10-5.

`-i`

Configure the library/repository for internal (operating system) file locking.

`-l`

Convert filenames to lowercase before processing.

`-m`

Compact the library by removing unused space. The original library is kept as `libname.old`. The compacted library will have read-write permissions and be configured to use internal file locking.`-l`

Convert filenames to lowercase before processing.

`-o`

Configure the library/repository to be read-only. This operation is not reversible unless the library is compacted using `formlib`'s `-m` option.

`-r`

Replace/add the named files to the library/repository.

`-s`

Synchronize the specified library with the source code management directory.

`-t`

Generate a list of the library/repository contents.

`-u`

Convert filenames to uppercase before processing.

`-v`

Display information in verbose mode. Generates a list of files processed. When used in conjunction with the `-t` option, produces a detailed listing of the library contents.

`-w`

Upgrade a JAM library/repository to Panther format.

`-z`

Attempt to recover deleted library members. Since the library code reuses the areas allocated to deleted library members before allocating new space, this operation should be performed as soon as possible after a member is deleted in error.

`filename`

Name of file to be included in library or repository.

`library`

Name of library/repository.

`memberName`

Name of library member or repository entries.

Description

`formlib` lets you create and maintain libraries/repositories in which you store application components, such as screens, menus, and JPL files. You can store ASCII files in a Panther library; however, only binary files are accessible at runtime or through the screen editor.

`formlib` can also be used to maintain and get information about libraries and repositories; for example, you can put a library/repository under source management control, or get a list of its contents.

File specifications can include any wildcard or pattern-matching symbols that are valid for your operating system. For example, this command puts all files with the `.rpt` extension into the library `screens.lib`:

```
formlib -c screens.lib *.rpt
```

Library member specifications must be explicit; no wildcards can be used.

Case Sensitivity

The `-l` and `-u` options are useful for operating systems like UNIX that are case-sensitive. For example, the following UNIX command creates the library `new.lib` and adds all `.scr` files in the current directory to it; all files receive lowercase names—for example, `MAIN.SCR` becomes `main.scr`.

```
formlib -cl new.lib *.[Ss][Cc][Rr]
```

File Locking

Access to libraries requires both read and write locking. Locking of libraries is unrelated to source control management, as the latter affects access to individual library members (and their archived versions) during development, whereas the former controls access on an as-needed basis to the library file as a unit during development and at runtime.

Libraries require write locks to provide exclusive access during the brief moments when a library is being written to. Read locks allow others to read from the library, but prevent the library from being written to while it is being read. Read-only libraries that have been marked so (via the `-o` option) do not need any locking at all since they cannot be written to under any circumstances.

You can configure a library to use either of two file-locking schemes. By default, the internal (native operating system) file locking system is used. Use the `-e` option to use an external locking scheme. External locking can be used if internal locking is not available in your environment. It creates an empty exclusive-access file called `libname.jlk` to implement write locking, and keeps a count of the current readers in a shared read lock file called `libname.rlk`. The `.jlk` and `.rlk` files are created in the same directory as the library. They are checked or modified when access to the library is attempted. The write lock file is deleted when access is completed; the read lock file is not. Also, the library is marked so that it knows to use external locking. Use the `-i` option to clear the external locking mark in the library, so that it will henceforth use internal locking.

Though external locking is portable, it has two drawbacks that make internal locking the preferable choice where available. First, it tends to be slower than internal locking. Secondly, if processes die abruptly—due to signals, operator intervention, and so on—write lock files can be left hanging around, or read locks can have counts that never go to zero. This impedes further access until a recovery procedure (stop all processes, delete the lock files) is performed. With internal locking, the operating system automatically performs clean up when processes die unexpectedly.

By default, libraries are created to use internal file locking. However, if a library is created on a platform where internal locking is not available, locking defaults to external. On such platforms, attempts to set internal locking are ignored. External locking might be necessary in cross-platform network environments where the network locking facilities are inadequate. Refer to your operating system's installation notes for further information about which file locking scheme is used on your platform and/or network environment.

Marking a Library Read-only

Use of the `-o` option makes a library read-only. This is not a reversible operation unless the library has been compacted using the `-m` option.

When the read-only operation is not reversible, in order to write to a read-only library, you have to create a new library, extract the members from the read-only library, and move them into the new library. Read-only libraries generally provide quicker access.

Synchronize Library with Source Code Management

Library copies of screens can become unsynchronized with source management directories. This can happen if screens are extracted and edited directly with the source code management tool as opposed to using the screen editor interface to check screens in and out.

Run `formlib` with the `-s` option to ensure that your libraries are synchronized with the source management directory. The specified library is moved to a new library with the same name, but with a `.jbl` (backup library) extension, and a new library is created, having the name of the original library. Any screens that were not under source code management are copied from the backup library to the new library. Screens that were under source management, but were checked out when `formlib` was executed are also copied from the backup library to the new library.

All screens that are under source code management and checked-in when `formlib` was executed are copied from the source code management directory to the new library. In this way, you can be assured that your libraries contain the latest revisions before distributing your libraries.

jif2asc

Converts JIF between binary and ASCII format for JetNet and Oracle Tuxedo applications

```
jif2asc -a [-fp]  ascJIF binJIF
jif2asc -b [-fp]  ascJIF binJIF
```

`-a`	Create ASCII JIF named *`ascJIF`* from specified binary *`binJIF`*.
`-b`	Create binary JIF named *`binJIF`* from specified *`ascJIF`*.
`-f`	Overwrite an existing file.
`-p`	Write the output file to the directory containing the input file.
`ascJIF`	Name of ASCII JIF to be converted or created.
`binJIF`	Name of binary JIF to be converted.

Description

JetNet and Oracle Tuxedo applications require a binary JIF for execution. In the JIF editor, the JIF is saved as binary. The `jif2asc` command-line utility lets you convert a JIF between ASCII and binary formats.

Both input and output files must be named, there is no default naming convention. The output file is created in the current directory, unless the -p option is used to indicate that the output file is to be written in the same directory as the input file.

An attempt to overwrite an existing file without using the `-f` option produces an error message and no file is written.

jpl2bin

Converts JPL modules between binary and ASCII formats

```
jpl2bin -a [ -fpv] [ -eext ] binary ...
jpl2bin -b [ -rfpv] [ -eext ] textFile ...
jpl2bin -s binary ...
```

`-a`
: Convert compiled JPL to source (default extension is *.`jpl`).

`-b`
: Compile JPL from source to binary format (default extension is *.`bin`).

`-e`*`ext`*
: Replace the default bin extension with the specified extension *`ext`* on the binary output filename. There should be no space between the -e switch and the extension name. To omit an extension, supply a value of `-` (dash) for *`ext`*; i.e. `-e-`.

`-f`
: Permit the output file to overwrite an existing file.

`-p`
: Output the binary file to the same directory as input file.

`-r`
: Exclude source from compiled binary. Useful for removing source from a production distribution. However, without the source, the module cannot be edited within the screen editor.

`-s`
: Strip source from the compiled binary (overwrites input).

`-v`
: List the name of each file as it is processed.

`binary`
: Name of the compiled JPL file.

`textFile`
: Name of the source text file.

Description

`jpl2bin` lets you compile JPL modules before storing them in a library or before making them memory-resident. Under UNIX, run `jpl2bin` from the command line; under Windows, choose the `jpl2bin` icon or run it from a command prompt; makefile or batch file.

Note: Panther always performs colon preprocessing at runtime; therefore, a module is fully compiled only when it executes.

`jpl2bin` saves the module to a file of the same name with a `*.bin` extension, unless you specify a different extension.

m2asc

Converts menu files between binary and ASCII formats

```
m2asc -a [ -fv] [ -i includeFile ] asciiFile menuFile [menuFile ...]
m2asc -b [ -fv] asciiFile [asciiFile ...]
```

`-a`
: Convert specified binary menu files to ASCII.

`-b`
: Convert the specified ASCII files to binary.

`-f`
: Overwrite an existing file.

`-i` *`includeFile`*
: Include specified `includeFile` at the beginning of ASCII output.

`-v`
: Generate a list of files as they are processed.

`asciiFile`
: Name of ASCII file as output with `-a` or as input file input with `-b`.

`menuFile`
: Name of binary menu file (as extracted with `formlib` utility).

Description

The `m2asc` utility lets you convert binary menu files to ASCII and vice versa. You must extract the menu file from its library with the `formlib` utility before using `m2asc`.

ASCII menu definitions define a menu as a hierarchy, where the top-level menu and its items are defined first along with global menu properties, followed by submenus and their items. Each component of a menu definition is identified by a keyword (refer to Table A-1 and, optionally, a unique name.

Table A-1 ASCII menu keywords

Menu keyword	Description
`ACTION`	Invokes an action through a control string.
`EDCLEAR`	Replaces the selected text with spaces.
`EDCOPY`	Copies selected text to the clipboard.
`EDCUT`	Cuts selected text to the clipboard.
`EDDEL`	Deletes the selected text.
`EDPASTE`	Pastes the clipboard contents.
`EDSELECT`	Selects the current widget's contents.
`FILE`	Source file of the menu script. You can write multiple menu scripts to the same ASCII text file; each script begins with a `FILE:script-name` identifier. When `m2asc` converts the ASCII file to binary format, each script is saved to its own file.
`MENU`	Starts a menu or submenu definition. All keywords that follow `MENU` identify the menu's items.
`SEP`	Draws a separator between the previous and next menu items.
`SUBMENU`	Invokes another menu. If the `SUBMENU` item is on the menu bar, the submenu displays as a pulldown; otherwise, the submenu displays to its right.
`TOGGLE`	Invokes an action through a control string and toggles the indicator on or off.
`WINLIST`	Identifies the item as a menu that lists all open windows.
`WINOP`	Identifies the item as the windows menu of the current plat form—for example, under Windows, the Windows menu with Arrange Icons, Tile, and Cascade. Applications running in character-mode ignore this item.

Each menu and menu item definition has properties; these properties are specified immediately below the component's identifier. For example, the following statements define a submenu item `myoption`: its label is `Options` with a keyboard mnemonic of `O`; it invokes the menu `myoptionsub`; and it is initially available for selection (`ACTIVE=YES`):

```
SUBMENU:myoption
  LABEL=&Options
  SUBMENU=myoptionsub
  ACTIVE=YES
```

Refer to Table 15-1 on page 15-15 for a list of menu-specific properties.

Example

The following menu script is the ASCII output of a truncated version of the menu bar used by Panther's screen editor. The example includes two of the main menu options and their associated submenus: `File` and `Help`.

```
FILE:semain

MENU:sm_se_main_menu
  ACTIVE=YES
  INDICATOR=NO
  SHOW-ACCEL=YES
  ACCEL-ACTIVE=YES

SUBMENU:sm_se_file
  LABEL=&File
  SUBMENU=sm_se_file_menu
  EXT-HELP-TAG=basicFilemenu
  STAT-TEXT=File Operations

SUBMENU:sm_se_help
  LABEL=&Help
  SUBMENU=sm_se_help_menu
  IS-HELP=YES
  STAT-TEXT=Get Help!

MENU:sm_se_file_menu
  TEAR=NO
  EXTERNAL=NO
  ACTIVE=YES
  INDICATOR=NO
  SHOW-ACCEL=YES
  SEP-STYLE=SINGLE

SUBMENU:sm_se_new
  LABEL=&New
  SUBMENU=sm_se_new_menu
  IS-HELP=NO
  EXT-HELP-TAG=FileNew
  STAT-TEXT=Create new screen

SUBMENU:sm_se_open
  LABEL=&Open
  SUBMENU=sm_se_open_menu
  IS-HELP=NO
  EXT-HELP-TAG=FileOpen
  STAT-TEXT=Open existing screen

ACTION:sm_se_save
  LABEL=&Save
  CONTROL=^jm_keys PF5
  ACTIVE=YES
  IS-HELP=NO
  EXT-HELP-TAG=FileSave
  ACCEL=PF5
  ACCEL-ACTIVE=NO
  SHOW-ACCEL=YES
  DISPLAY-ON=BOTH
  STAT-TEXT=Saves the current screen
  ORDER=18
  ACTIVE-PIXMAP=save-act
  INACTIVE-PIXMAP=save-dis
  TOOL-TIP=Save

ACTION:sm_se_set_test
  LABEL=&Test Mode
  CONTROL=^jm_keys PF2
  IS-HELP=NO
  EXT-HELP-TAG=FileTestMode
  ACCEL=PF2
  ACCEL-ACTIVE=NO
  SHOW-ACCEL=YES
  DISPLAY-ON=BOTH
  STAT-TEXT=Switch to Test Mode
  ORDER=19
  ACTIVE-PIXMAP=test-act
  TOOL-TIP=Test Mode

SEP:sm_se_file_sep2
  SEP-STYLE=SINGLE

ACTION:sm_se_exit
  LABEL=E&xit
  CONTROL=^jm_keys CLAPP
  ACCEL=CLAPP
  IS-HELP=NO
  EXT-HELP-TAG=FileExit
  ACCEL-ACTIVE=NO
  SHOW-ACCEL=YES
  STAT-TEXT=Exit the editor

MENU:sm_se_new_menu
  TEAR=NO
  EXTERNAL=NO
  ACTIVE=YES
  INDICATOR=NO
  SHOW-ACCEL=YES
  SEP-STYLE=SINGLE

ACTION:sm_se_new_screen
  LABEL=&Screen
  CONTROL=^filemenu new screen
  IS-HELP=NO
  EXT-HELP-TAG=FileNew
  SHOW-ACCEL=YES
  DISPLAY-ON=BOTH
  STAT-TEXT=Creates new untitled screen
  ORDER=11
  ACTIVE-PIXMAP=new-act
  INACTIVE-PIXMAP=new-dis
  TOOL-TIP=New Screen

ACTION:sm_se_new_jpl
  LABEL=&JPL
  CONTROL=^filemenu new jpl
  IS-HELP=NO
  EXT-HELP-TAG=FileNew
  SHOW-ACCEL=YES
  STAT-TEXT=Creates new jpl

ACTION:sm_se_new_dd_entry
  LABEL=Repository &Entry...
  CONTROL=^filemenu new ddentry
  IS-HELP=NO
  EXT-HELP-TAG=FileNew
  SHOW-ACCEL=YES
  STAT-TEXT=Create new repository entry

ACTION:sm_se_new_lib
  LABEL=&Library...
  CONTROL=^filemenu new lib
  IS-HELP=NO
  EXT-HELP-TAG=FileNew
  SHOW-ACCEL=YES
  STAT-TEXT=Create a new library

MENU:sm_se_open_menu
  TEAR=NO
  EXTERNAL=NO
  ACTIVE=YES
  INDICATOR=NO
  SHOW-ACCEL=YES
  SEP-STYLE=SINGLE

ACTION:sm_se_op_lib
  LABEL=&Library...
  CONTROL=^filemenu open lib
  IS-HELP=NO
  EXT-HELP-TAG=FileOpen
  SHOW-ACCEL=YES
  STAT-TEXT=Open library

ACTION:sm_se_op_db
  LABEL=D&atabase...
  CONTROL=^dm_handle_connect 1
  ACTIVE=NO
  IS-HELP=NO
  EXT-HELP-TAG=FileOpen
  SHOW-ACCEL=YES
  STAT-TEXT=Open database

MENU:sm_se_help_menu
  TEAR=NO
  EXTERNAL=NO
  ACTIVE=YES
  INDICATOR=NO
  SHOW-ACCEL=YES
  SEP-STYLE=SINGLE

ACTION:sm_se_hl_topic
  LABEL=Current &Topic ...
  CONTROL=^jm_keys HELP
  IS-HELP=NO
  STAT-TEXT=Shows help on what you're doing
  ORDER=191
  DISPLAY-ON=BOTH
  ACTIVE-PIXMAP=help-act
  INACTIVE-PIXMAP=help-dis
  TOOL-TIP=Help

SEP:sm_se_hl_sep1
  SEP-STYLE=SINGLE
  ORDER=190
  DISPLAY-ON=BOTH

ACTION:sm_se_hl_about
  LABEL=&About Panther ...
  CONTROL=^sm_message_box( \
  "Panther Version 5.5%NCopyright 1994-2016%NProlifics Inc.", \
  "About Panther",0,"")
  STAT-TEXT=Tells about this version of Panther
```

msg2bin

Converts ASCII message files to binary format

```
msg2bin [-pv] [-e ext] msgFile...
msg2bin [-pv] [-o file] msgFile...
```

-e *ext*
: Replace default `bin` extension on the output file with the given extension (*ext*). If the `-o` option is used, `-e` is ignored.

-o *file*
: Output is placed in a single specified file. Use this option to concatenate your user messages to Panther-messages in a single binary file. This option will overwrite an existing binary message file of the same name.

`-p`
: Place each binary output file in the same directory as the corresponding input file.

`-v`
: List the name of each input message file as it is processed.

msgFile
: Name of ASCII file containing named messages. More than one input file can be specified.

Description

`msg2bin` converts ASCII message files to a binary format for use by Panther library functions. The output of `msg2bin` is a binary file; the utility uses the `TAGs` to distinguish between system and user messages. It numbers user-defined messages consecutively starting with the class number times `0x1000` (see page 45-7 for more about defining user message classes). If no classes are defined, user-defined messages are automatically numbered consecutively starting from zero; the definitions of system messages are taken from `smerror.h`. Be sure to maintain the order of messages and the assignment of their identifiers. Use these identifiers in the application programs to select the desired messages at runtime. Then recompile and link any non-JPL source that includes any files that contain newly defined messages.

Errors

The following table describes possible errors, their cause, and the corrective action to take:

At least one file name is required.

Cause	No message file was specified.
Action	Specify the name of the message file.

Error in %s line %d: bad tag %s
Warning in %s line %d: bad tag %s

Cause	The tag is missing or does not consist entirely of letters, digits and/or underscores. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending tag and rerun `msg2bin`. Refer to `smerror.h` for a list of tags.

Error in %s line %d: duplicate message tag %s
Warning in %s line %d: duplicatemessage tag %s

Cause	An earlier line also contained the same system message tag. The current line is skipped. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending line and rerun `msg2bin`.

Error in %s line %d: duplicate user section class %s
Warning in %s line %d:duplicate user section class %s

Cause	The user section or two letter prefix on a class indicator line is already in use. Class zero is implicitly used if a user message is encountered before any class indicator lines. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending line and rerun `msg2bin`.

Error in %s line %d: invalid user message class indicator line %s
Warning in %sline %d: invalid user message class indicator line %s

Cause	A user class indicator line (which is used to start a new message User Section) is defective. The programs assume that any tag starting with a double quote is one of these. The tag must be a double quote followed by two character alphanumeric code and a double quote. The class indicator must be a digit between `0` and `7`. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending line and rerun `msg2bin`..

Error in %s line %d: message tag exceeds 80 characters %s
Warning in %s line%d: message tag exceeds 80 characters %s

Cause	Message name too long.
Action	Shorten the message name and rerun `msg2bin`.

Error in %s line %d: missing final quote &s
Warning in %s line %d: missing final quote &s

Cause	The message content starts with a quote character (`"`, `'`, `` ` ``) but does not contain a matching terminal quote character. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending message and rerun `msg2bin`.

Error in %s line %d: text after final quote %s
Warning in %s line %d: text after final quote %s

Cause	The message content starts with a quote character (`"`, `'`, `` ` ``) but there are characters after the matching terminal quote character. Perhaps a backslash is missing. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending line and rerun `msg2bin`.

Error processing file %s

Cause	An error was encountered reading or writing the file.
Action	Confirm that the file is available and that the target directory can be written.

File '%s' not found.

Cause	An input file was missing or unreadable.
Action	Check the spelling, presence, and permissions of the file in question.

Insufficient memory available.

Cause	The utility could not allocate enough memory for its needs.
Action	None.

Invalid character(s) in -x option.

Cause	The `-x` option (characters to prefix to the tag) starts with a digit or contain a character that is not an alphanumeric or an underscore.
Action	Specify a valid prefix and rerun `msg2bin`.

Line too long: %s

Cause	The message has exceeded `1024` characters.
Action	Split the message into two separate messages or edit the message length.

Message tag exceeds 80 characters:%s

Cause	Message name too long.
Action	Shorten the message name and rerun `msg2bin`.

Missing '=' in line: %s

Cause	The line in the message had no equal sign following the tag.
Action	Correct the input and rerun `msg2bin`.

Warning in %s line %d: message tag exceeds 31 characters %s

Cause	The tag is longer than 31 characters but is shorter than 80.
Action	None.

Warning in %s line %d: prefix does not match user section class %s

Cause	The first two characters of a user message do not match the two characters specified in the most recent user message class indicator line. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending tag and rerun `msg2bin`.

Warning: no messages in user section \"%.2s\"

Cause	The input file had a user message class indicator line that did not have any user messages after it. That section will not be included in the output file.
Action	Remove the offending class indicator and rerun :`msg2bin`.

msg2hdr

Creates header files for user messages

```
msg2hdr [-dfjpv] [-n num] [-s pfix] [-x pfix] [-o file] [-e ext]
    msgFile...
```

`-d`
: Decimal base in the output header file. Default is base `16` (hexadecimal).

`-e` *`ext`*
: Replace default extension (`h` or `jpl`) on the output file with the specified extension *`ext`*.

`-f`
: Output file may overwrite an existing file.

`-j`
: Create a JPL global variable file from *`msgFile`*.

`-n` *`num`*
: Start numbering messages with the specified num for the first #define or global. If no number is entered, `0` (zero) is used.

`-o` *`file`*
: Direct output to the named file.

`-p`
: Place output file in the same directory as the corresponding input file.

`-s` *`pfix`*
: Select only message names beginning with the specified prefix *`pfix`*.

`-v`
: Generate list of the files processed.

`-x` *`pfix`*
: Prepend the specified prefix *`pfix`* to the tag portion of the message.

`msgFile`
: Name of ASCII file containing your application's messages.

Description

msg2hdr converts an ASCII message file that contains your application's messages to a C header file. The output of msg2hdr is a .h file with #define statements for each user message tag. The messages are numbered sequentially starting with 0x0 to 0xF. The message portion is copied to the header file as a comment.

For example, a user message file with multiple sections might look like this:

```
"U0" = 0
U0_BADVAL    =  Bad value
U0_WRONGDATE =  Date must be within 30 days of current date
"U1" = 1
WRONGRATE    =  This is not the applicable rate
```

This yields the following output:

```
#define U0_BADVAL      0x0     /* Bad value             */
#define UO_WRONGDATE   0x1     /* Date must be within 30 \
>                                 days of current date */
#define WRONGRATE      0x1000  /* This is not the \
>                                 applicable rate */
```

```
If you use the -j option, msg2hdr yields this output:
```

```
global U0_BADVAL(1)    = 0     /* Bad value             */
global U0_WRONGDATE(1) = 1     /* Date must be within 30 \
>                                 days of current date  */
global WRONGRATE(4)    = 4096  /* This is not the \
>                                 applicable rate */
```

Errors

The following table describes possible errors, their causes, and the corrective action to take:

At least one file name is required.

Cause	No message file was specified.
Action	Specify the name of the message file.

Error in %s line %d: bad tag %s
Warning in %s line %d: bad tag %s

Cause	The tag is missing or does not consist entirely of letters, digits and/or underscores. It can also indicate the first character of the tag is a digit. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending tag and rerun `msg2hdr`. Refer to `smerror.h` for a list of tags.

Error in %s line %d: duplicate message tag %s
Warning in %s line %d: duplicatemessage tag %s

Cause	An earlier line also contained the same system message tag. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending tag and rerun `msg2hdr`.

Error in %s line %d: duplicate user section class %s
Warning in %s line %d:duplicate user section class %s

Cause	The user section or two letter prefix on a class indicator line is already in use. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending line and rerun `msg2hdr`.

Error in %s line %d: invalid user message class indicator line %s
Warning in %sline %d: invalid user message class indicator line %s

Cause	A user class indicator line (which is used to start a new message User Section) is defective. The programs assume that any tag starting with a double quote is one of these. The tag must be a double quote followed by two character alphanumeric code and a double quote. The class indicator must be a digit between `0` and `7`. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending line and rerun `msg2hdr`.

Error in %s line %d: message tag exceeds 80 characters %s
Warning in %s line%d: message tag exceeds 80 characters %s

Cause	The message name (tag) is longer than 80 characters.
Action	Shorten the message name and rerun `msg2hdr`.

Error in %s line %d: message tag exceeds 31 characters %s
Warning in %s line %d: message tag exceeds 31 characters %s

Cause	The tag is longer than 31 characters but is shorter than 80. This causes an error if the -j (JPL) option is selected, otherwise it causes a warning. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending tag and rerun `msg2hdr`.

Error in %s line %d: prefix does not match user section class
Warning in %s line %d: prefix does not match user section class %s

Cause	The first two characters of a user message do not match the two characters specified in the most recent user message class indicator line. An output file will be created and `RET_SUCCESS` will be returned if warnings are encountered but no errors.
Action	Fix the offending tag and rerun `msg2hdr`.

Error processing file %s

Cause	An error was encountered reading or writing the file.
Action	Confirm that the file is available and that the target directory can be written.

Exactly one message file name is required.

Cause	More than one input message file was specified.
Action	Run `msg2hdr` separately for each message file. Consider using the `-n` option on the subsequent messages to number the messages consecutively.

File '%s' already exists; use '-f' to overwrite.

Cause	An output file of the same name already exists.
Action	Use the `-o` option to specify a different output name or use the `-f` option to overwrite the existing header file.

If no number is entered, 0 will be used.

Cause	You did not provide a number with the `-n` option and it defaulted to zero.
Action	Rerun `msg2hdr` providing a number.

Insufficient memory available.

Cause	The utility could not allocate enough memory for its needs.
Action	None.

Invalid all-numeric message name '%s'

Cause	At least one non-numeric character must be in a message name.
Action	Rename the offending message and rerun `msg2hdr`. (If you have already converted the message file to binary, you will need to rerun `msg2bin`.)

Invalid character(s) in -x option.

Cause	The `-x` option (characters to prefix to the tag) starts with a digit or contain a character that is not an alphanumeric or an underscore.
Action	Specify a valid prefix and rerun `msg2hdr`.

Missing '=' in line: %s

Cause	The line in the message had no equal sign following the tag.
Action	Correct the input and rerun `msg2hdr`. (If you have already converted the message file to binary, you will need to rerun `msg2bin.`)

Missing message name for '%s'

Cause	The message had no characters before the equal sign.
Action	Provide a name for the offending message and rerun `msg2hdr`. (If you have already converted the message file to binary, you will need to rerun `msg2bin`.)

Warning: no messages in user section \"%.2s\"

Cause	The input file had a user message class indicator line that did not have any user messages after it.
Action	Remove the offending class indicator and rerun `msg2hdr`.

s2asc

Converts styles files between binary and ASCII formats

```
s2asc -a [-f] asciiFile styles.sty
s2asc -b [-f] asciiFile
```

`-a`
: Create ASCII listing of `styles.sty`. The styles file must be named `styles.sty`.

`-b`
: Create a binary style file from an ASCII listing. The name of the binary file corresponds to the name specified in the `s:name` option in the ASCII file. To have this binary file accessed by your application, change the name of the file to `styles.sty`.

`-f`
: Overwrite an existing `styles.sty` file.

`asciiFile`
: With the `-a` option, name of the file in which to place the ASCII styles settings. With the `-b` option, name of the file containing ASCII text for conversion to binary format.

Description

The `s2asc` utility must be executed from the command line. With the utility, you can convert your `styles.sty` file between binary and ASCII format. After converting an ASCII file to binary with `s2asc`, use `formlib` to put it in the appropriate library. You might do this in order to place the file under source control management or to document or review the contents of the file.

The text files generated by `s2asc` contain a list of the property settings for each style, followed by a list of the styles assigned to each class.

APPENDIX

B VideoBiz

VideoBiz is a sample application that gives you a look at a working two-tier application. Although all Panther features are not implemented in this application, it is designed to illustrate some of the functionality that is possible. In addition, VideoBiz can help point out how some things can be done—and can assist you in developing your two-tier application.

VideoBiz is a database application that was built to take advantage of Panther's transaction manager capabilities. The application required very little coding—only a minimal amount of SQL and JPL code. The application screens were constructed via the screen wizard or by using widgets imported from the underlying database and inherited from a repository. Properties of these widgets, along with those of table views and links, provide the information that the transaction manager needs to drive the automated SQL generation.

This chapter describes:

- Starting VideoBiz—The start up information you need to actually view and use the VideoBiz application.
- What You Get with VideoBiz—A brief description of the components that make up this application, such as the menu bar, the JPL code, and the screens that comprise the application.
- The User's Guide to VideoBiz—The functional specification to the user interface which provides a "how-to-use" approach from a user's perspective as well as some insight into what's happening in the background. This section walks you through each screen and briefly describes how the screen works and what actions the user can take.

  Check out the "Behind the Scenes" section for each screen. These sections outline what features are implemented, where you can find the JPL code, and what mechanisms are used to make a behavior or event occur.

Starting VideoBiz

To look at VideoBiz follow the directions below for your specific platform and environment. Try the application (refer to page B-8 for more information on using the application). In the process of looking at VideoBiz, you can also invoke the screen editor to look behind the scenes to find out just how it works.

While VideoBiz is running, you are in Application mode. To see how the current application screen looks behind the scenes, you must be in Edit mode. To do this, simply access the screen editor by choosing panther→Screen Editor from the application menu bar. The current screen will open in the screen editor workspace. In this way, you can see all the property specifications for the screen and its widgets, look at the JPL code that is attached to the specific screen, and see *all* the widgets on the screen, including those that are hidden at runtime, like table view links. To return to Application mode, choose File→Exit from the screen editor menu bar.

How to Start VideoBiz

For Windows:

To start VideoBiz, choose the VideoBiz icon in the Panther program group. The VideoBiz Welcome screen opens.

For Motif and Character- Mode:

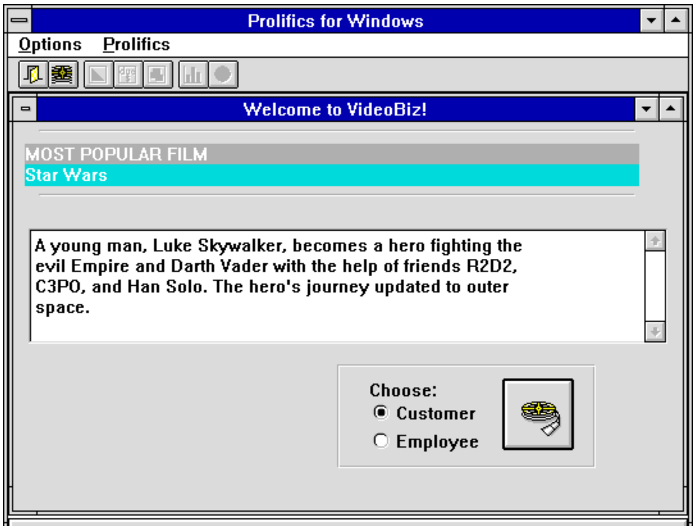
1. Know in what directory Panther is installed.
2. From your home or working directory (make sure the directory has write-per missions), run the following script:

   ```
   $SMBASE/samples/videobiz/vbizunix
   ```

   The script copies the required files to your current directory. They include the `videobiz` database and the styles file `styles.sty`.

3. When prompted, enter `c` or `m` to indicate whether you are using VideoBiz on a color or monochrome monitor.

   The VideoBiz Welcome screen opens.



MOST POPULAR FILM
Star Wars

A young man, Luke Skywalker, becomes a hero fighting the evil Empire and Darth Vader with the help of friends R2D2, C3PO, and Han Solo. The hero's journey updated to outer space.

Choose: [✓ Customer] Employee

Figure B-1 VideoBiz Welcome screen includes a menu bar, a graphical toolbar, two logon options, and a display of the most frequently rented video.

VideoBiz Components

This section describes the contents of the `$SMBASE/samples/videobiz` directory and how VideoBiz uses these elements. These include:

- The `videobiz` database
- Repository (supplied for reference; this is not runtime requirement)
- Application screens
- Menu bar/toolbar

- JPL code
- Styles sheet
- Sample reports
- Pixmap files

The Database

VideoBiz runs against a JDB relational database, `videobiz`, that has been normalized. The primary and foreign key definitions were made in the SQL table creation statements. The VideoBiz application depends on these definitions to drive the transaction manager's access to the SQL generator.

The Repository

The repository `data.dic` that was created and used to build VideoBiz is provided so that you can see what kinds of things are controlled via this mechanism. Panther's visual object repository and its inheritance mechanism is a development tool used to implement and maintain application consistency, to store reusable application components, and to facilitate application maintenance as well as provide the screen wizard with the information it needs to quickly create screens. This repository contains three general types of screens: those created as the result of the database tables import process, those created and used by the screen wizard, and one that was created simply as a screen to hold frequently used objects, like push buttons.

Imported Database Tables

By importing database tables as repository screens, the widgets that were derived from the database were used to build the VideoBiz application screens. Attributes of the source database table are embedded in the corresponding widget's Database properties. These properties are inherited by child copies of these widgets, and provide the SQL generator with the information needed to dynamically generate SQL statements. Changes in the underlying database tables can be re-imported into the repository and then inherited by the child screens and widgets.

Many properties imported from the database do not exactly correspond to your application's business requirements. For example, database tables imported from JDB automatically assign a Length property of 11 to widgets imported from database columns of type long. In VideoBiz, an ID number is never longer than 5 characters. This was resolved, for example, by changing the Length property of the `cust_id` widget from 11 to 5 in the repository. This ensures that wherever that widget is used in the application, its length is 5 characters long.

Other properties that could reasonably be changed at the database table level were considered, and some implemented. The things to consider are:

- Does it make sense to propagate the change to every child copy in the application?
- Is it useful to control the settings of this property with inheritance?

Some of the property changes made in the repository include input keystroke filters, the Length property, font specifications for data entry widgets, and data format specifications (for instance, date formats).

Other Repository Entries

Repository-based inheritance was also used to define standard widget types (in this case, push buttons). These are stored in the repository screen `masters.wgt`. The appearance of the push buttons on the application screens is inherited from this repository screen. In this manner, a consistent look is propagated and easily maintained.

Wizard entries (`smwizard` and `smwizis`) were automatically created when the screen wizard was first invoked.

Application Screens

The VideoBiz screens were created by using the screen wizard. The wizard uses the database-derived widgets in the repository to build screens.

While you navigate through the VideoBiz application, you can also examine what’s going on behind the scenes (or screens) by invoking the screen editor (choose panther→Screen Editor). The current screen will open in the screen editor workspace and you can see how the screen was put together, what properties were set, and which properties are inherited from the repository (these are displayed in reverse video in the Properties window).

When you are done, resume the VideoBiz application by choosing File→Exit from the screen editor menu bar.

Menu Bar/Toolbar

The menu bar and toolbar in VideoBiz is used primarily for navigation among the modules. Panther's menu bar editor was used to create the menu script file. The menu script is read into memory when the Welcome screen opens. It remains in place for the life of the application.

Items on the Options menu become active or inactive depending on the user's permissions and the currently active screen. For example, a user with customer permissions will not be allowed to run reports or view customer profiles.

A Panther menu option is also provided on the menu bar to allow you to easily access the Panther editors and to view the SQL that is being automatically generated for the VideoBiz application. This menu bar item would normally not be part of a distributed, runtime-only application, but is provided for your convenience.

JPL Code

All of the coding in VideoBiz is done with JPL and is well-documented. There is one externally stored file, `videobiz.jpl`, which is called when the first screen (`main.jam`) is opened. All the procedures contained in this file are then globally available to the application. This is particularly useful when a procedure is used by more than one screen. For example, the procedure `init_menu` is used throughout the application.

All other JPL code is stored with the screens that use it.

Styles Sheet

The default Panther `styles.sty` file was modified to accommodate the VideoBiz application. This file controls how widgets behave when different transaction modes are executed.

Sample Reports

The Marketing portion of VideoBiz generates reports. If your Panther executable includes ReportWriter, edit the `videobiz.jpl` file to let Panther know this. Change the following line:

```
RW_INSTALL = 0
```

to

```
RW_INSTALL = 1
```

The report templates are provided as Panther screens by the ReportWriter installation. You can see how they are constructed by opening them in the screen editor; they are: `duenote.jam`, `topten.jam`, and `genrecus.jam`.

Pixmap Files

There are several pixmap files provided that serve to enhance the VideoBiz application on GUI platforms. There are pixmap files used for push buttons, toolbar items, for the screen when it is minimized, and for screen wallpaper.

The User's Guide to VideoBiz

This section serves as a functional specification for the VideoBiz user interface. The specification is usually where an application begins. A task is introduced and a solution is sought. The user's perspective introduces you to what VideoBiz is intended to do and what functions it will perform.

What is VideoBiz?

VideoBiz is a small database application that is intended for use in a video rental store. It serves three audiences and provides functions specific to those audiences:

- Customers—to look up information about videos.
- Front desk clerks—to add and change information about customers, to look up video information, to rent videos and check them back in.
- Marketing personnel—to produce reports about customers and the videos they rent.

Starting VideoBiz

VideoBiz runs on free-standing terminal kiosks in the video store for customer use as well as on work stations behind the front desk for employee use.

The Welcome (`main.jam`) screen displays when the application is idle, that is, when no one has logged on, and it shows the title and description of the most frequently rented video.

The application's menu bar offers two menu bar items: Options and Panther. The toolbar includes all entries available via the Options menu. On initialization, the only available choices under Options and on the toolbar are Video Search and Done/Exit. The user can choose Video Search to search for a video by title ID, title, and/or director.

The Panther menu is provided for your convenience to give you access to the Panther editors so that you can examine the internals of the application. This option would normally not be part of a runtime application.

The Welcome screen also includes two radio buttons:

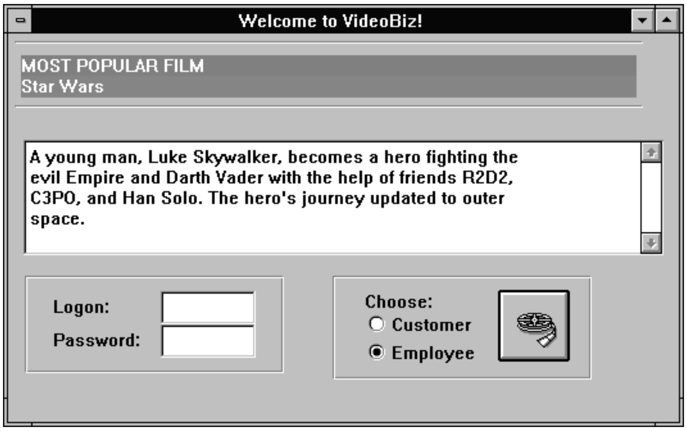
- Customer—The default radio button. When this button is selected, pressing Enter or choosing the Start button invokes the Search for a Video screen. (Under GUI platforms, the Start button displays a pixmap of a 35mm camera reel on it.)

  No log on information is required. The application navigates to the Video Lookup screen when the Start button is chosen. Video Search and Done are the only available menu options when the screen is in Customer mode.

- Employee—Requires the user to enter a user name and password. When this button is selected and the required login data is provided, pressing Enter or choosing the Start button invokes the Customer Search screen.

How to Log into VideoBiz as an Employee

1. Choose the Employee radio button. The log on fields (Name and Password) are displayed.



Welcome to VideoBiz!

MOST POPULAR FILM
Star Wars

A young man, Luke Skywalker, becomes a hero fighting the evil Empire and Darth Vader with the help of friends R2D2, C3PO, and Han Solo. The hero's journey updated to outer space.

Logon:
Password:

Choose:
Customer [✓ Employee]

Figure B-2 The Welcome screen displays logon and password fields for employee access.

2. Enter the name `sheila` in the Name field, and `trade3` in the Password field. (The password is echoed using asterisks (*).)

   Both user name and password are required. An error message is posted if both are not provided. Otherwise, logon information is compared to a list of valid names and passwords. An invalid user name or password invokes an error message and the user can try another.

3. Choose the Start button (or press Enter). The Search for a Customer screen opens.

Connection to the database occurs upon initial screen entry. On entry, the screen is by default in Customer mode. If a valid user/password is entered, the application switches into Employee mode. While in Employee mode, the marketing menu/ toolbar items are active. When a user returns to the Welcome screen, the application automatically switches back to Customer mode.

How to Exit VideoBiz

Choose Close/Quit from the application's system menu. Panther prompts you to confirm the termination of your session in VideoBiz.

Behind the Scenes

The `main.jam` screen includes the following features which you can examine by accessing the screen editor:

- When the screen opens it calls `videobiz.jpl` to install the application menu and JPL procedures, making them globally available to the application.
- Call to `init_menu` to turn on the applicable menu/toolbar selections.
- Silent connection to and disconnection from the `videobiz` database. Check out the JPL Procedures property for the screen.
- The Name and Password fields are hidden in Customer mode and exposed conditionally. This is controlled via the screen-level JPL (the procedure name is `display_login_fields`). The screen-level JPL also handles the validation of the user name and password.
- A pixmap (visible under GUI platforms) is attached to the Start button via its Active Pixmap property under Format/Display.

- An icon visible on GUI platforms displays when the screen is minimized. It is defined in the screen's Icon property.
- Three database tables are linked on this screen so that the most popular movie title and description are displayed. Check the DB Interactions window in the screen editor to see how the table views are linked.
- The `sm_tm_command(“VIEW”)` in the screen-level JPL invokes the transaction manager to execute the query that determines the most frequently rented video. See the Database properties for the hidden widget `times_rented`. This widget provides the aggregate expression used in the SQL that is automatically generated on screen entry.

Identify the Customer

When the user successfully logs in as an employee, the Search for a Customer (`custlist.jam`) screen opens.

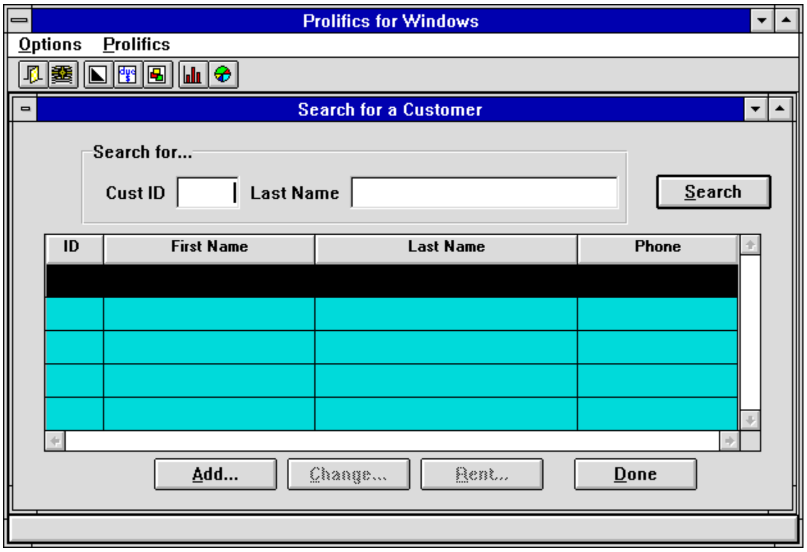
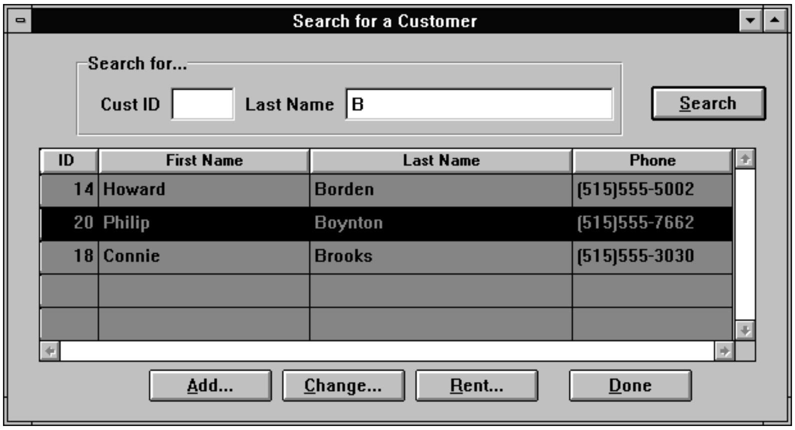


Figure B-3 Customer search screen allows employee users to search for specific customers.

This screen allows the employee to search for an individual customer and select an action for that customer. The screen includes two query fields and a grid widget which displays the results of the query—the customer's ID, first and last names, and phone number. A bounce bar can be moved up and down in the grid to indicate the currently selected customer. The screen includes several push buttons and Customer Profile menu bar/toolbar options that invoke other screens.

The top portion of the screen provides two fields on which the user can query. Customer records can be searched in two ways: by customer ID or by full or partial last name.



Cust ID:
Last Name: B

ID	First Name	Last Name	Phone
14	Howard	Borden	(515)555-5002
20	Philip	Boynton	(515)555-7662
18	Connie	Brooks	(515)555-3030

Figure B-4 Search for a specific customer record using an ID number or last name.

How To Search for a Customer Record

1. Specify the search criteria by doing one of the following:
   - Enter a customer ID in the Cust ID field at the top of the screen, and choose the Search button (or press Enter).

     If there is a corresponding customer record, the data are displayed in the grid widget.
   - Enter a full or partial string in the Last Name field at the top of the screen, and then choose the Search button (or press Enter).

All records that match the search criteria are displayed in the grid widget. For example, if the user enters B into the Last Name field, all customers whose last names begin with B are displayed.

- Choose the Search button (the screen’s default button) or press Enter.

  All customer records are fetched, and are displayed in alphabetical order in the grid widget.

Choosing the Search push button or pressing Enter (activates the screen’s default button) triggers a search using the contents of the query fields as criteria. At any point, the user can enter a new search string or customer ID in the query fields and trigger a new search by selecting the Search button (or pressing Enter).

If no records match the search criteria, the application prompts the user to add a new customer. If the user chooses Yes to add a new customer, the Customer Information screen opens. If the user chooses No, new search criteria can be specified.

2. Select the desired record by doing any of the following:

   - Click in any cell of the grid widget to highlight the record.
   - Press the up and down arrow keys to position the bounce bar on the desired record.
   - Double-click in any cell of the grid widget to select the record and invoke the Customer Information screen. The selected customer record is displayed and ready for update.

   The selected record is the target of any commands triggered by selecting a push button or one of the Customer Profile menu bar/toolbar item (pie or bar chart).

3. Specify the kind of action to take by doing any of the following:

   - Choose the Add button—Ignores any search criteria and opens the Customer Information screen. The application is ready to insert a new customer record (i.e., the screen is in New mode) into the database.
   - Double-click on a customer record or choose the Change button—The Change push button is inactive, or grayed, if a customer record is not selected. On a customer record is selected and the Change option is armed, the Customer Information screen opens and displays details of the selected customer record. The application is ready to update this record (i.e., the screen is in Update mode).

- Choose the Rent button—The Rent push button is inactive, or grayed, if a search has not been conducted; that is, if the grid widget displays no data. The button becomes active once a customer record is selected in the grid widget. It navigates to the Video Rentals screen where the selected customer's current video rentals are displayed.
- Choose the Bar Chart/Pie Chart button or Options→Profile→Bar Chart or Pie Chart—Brings up the customer's rental profile in the specified `chart_type` format.
- Choose the Done button or Options→Done—Closes the Search for a Customer screen and returns to the Welcome screen.

Behind the Scenes

The `custlist.jam` screen includes the following features which you can examine by accessing the screen editor:

- ■ Query by example allows the user to enter search criteria. The Use In Where Operator property for query fields defines what data to fetch that matches the search criteria.
- ■ The menu/toolbar item's and push button's active/inactive behavior is controlled by a style defined in the `styles.sty` file. See the Class property (under Transaction) for the Rent and Change buttons.
- ■ The Control String property under Validation for each of the push buttons on this screen controls how each button behaves when it is selected.
- ■ The Customer Profile options are activated on screen entry, but the code can be changed easily to have them activated only when a customer is selected.
- ■ The Double Click property is set on each of the grid members.
- ■ The telephone number format is controlled by a JPL procedure that is invoked as a validation function on the phone column grid member. The Force Valid property (under Database) forces the validation when data are selected into this field.
- ■ The customer search procedure (`custsearch`) is invoked from the Search push button. The procedure is stored as screen-level JPL.
- ■ The screen-level JPL includes code to determine what should happen when a customer query returns no records.

Add/Update Customer Records

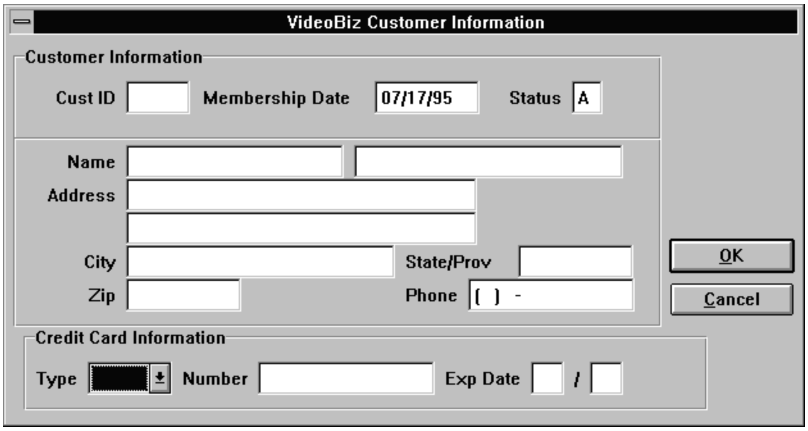
The Customer Information (`custedit.jam`) screen displays detailed information about a selected customer. The user arrives at this screen for one of two reasons: to add or to change a customer record. Once the record is inserted or updated, choosing OK commits the additions/changes.

The Cancel button closes the Customer Information screen without saving any changes, and returns the user to the customer search screen.

How to Insert a Customer Record

1. Choose the Add push button on the Search for a Customer (`custlist`) screen or respond to the application prompt to add a new customer (when a query results in no matches).

   The Customer Information screen opens in New mode.



Membership Date 07/17/95
Status A
Phone ( ) -

Figure B-5 Customer Information in New mode initializes the Membership Date to the system date and the Status field to A, for active. Also, any string that the user entered in Last Name query field on the Search for a Customer screen is passed to this screen and is displayed in the Last Name field.

2. Tab or click to each field and enter the customer demographics and credit card information.

The Cust ID, Membership Date and Status fields are protected from input. All other fields are ready for data entry.

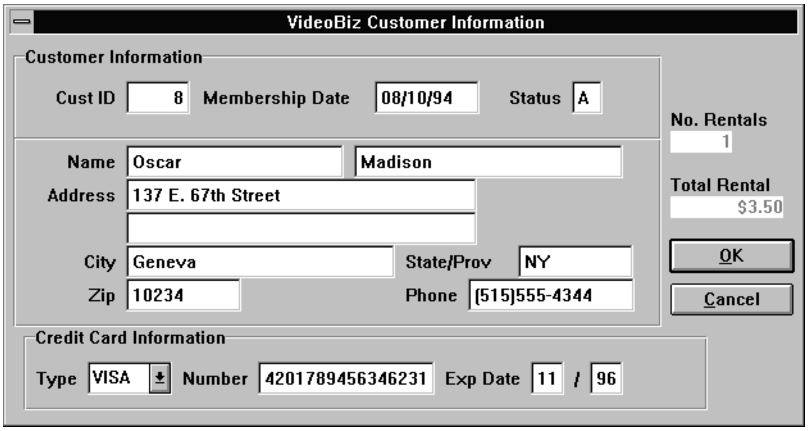
3. Choose OK to accept the information. The application assigns a customer ID number and displays a confirmation message.

   The OK button executes a procedure to assign a customer identification number and to insert (save) the new record to the database because a New command was specified. Once the confirmation message is acknowledged, the screen closes, and the user returns to the customer search screen.

How to Update a Customer Record

1. Select a customer in the grid and choose the Change button or double-click on a customer record on the Search for a Customer (`custlist`) screen.

   The Customer Information (`custedit`) screen opens.



Cust ID	8
Membership Date	08/10/94
Status	A
Name	Oscar Madison
Address	137 E. 67th Street
City	Geneva
State/Prov	NY
Zip	10234
Phone	(515)555-4344
No. Rentals	1
Total Rental	$3.50
Type	VISA
Number	4201789456346231
Exp Date	11 / 96

Figure B-6 The Customer Information screen displays data associated with the selected customer record.

2. Tab or click to the fields that require change.

   All fields are updatable, except for the customer ID and the rental information fields.

3. Choose OK to commit the changes to the database. An update confirmation message is displayed.

When updating an existing record, the OK button performs an update of the database. Once the confirmation message is acknowledged, the screen closes, and the user returns to the customer search screen.

Behind the Scenes

The `custedit.jam` screen includes the following features which you can examine by accessing the screen editor:

- Receives a transaction manager command and data from the Search for a Customer (`custlist.jam`) screen and performs a `COPY` or `SELECT` command, depending on what action the user has specified.
- The Credit Card option menu is populated from the database via a screen called `credcard.jam`. Look at the Identity properties for the option menu to see how this lookup capability is implemented. Then open the `credcard.jam` screen in the screen editor to see how selection screens are constructed. It's the `credcard` screen's entry function that actually runs the query to fetch credit card information.
- The customer ID is provided programmatically when a new customer record is created. Review the screen-level JPL procedure (`ok_proc`) attached to this screen.
- Fields are updatable based on the transaction class that the widget is assigned to–look at the Class property under Transaction for the `cust_id` and `member_date` text widgets.

Video Rental Listing

Users with “front desk” permissions can reach the Video Rental Listing screen by first identifying and selecting a customer record from the Search for a Customer screen and then choosing the Rent push button.

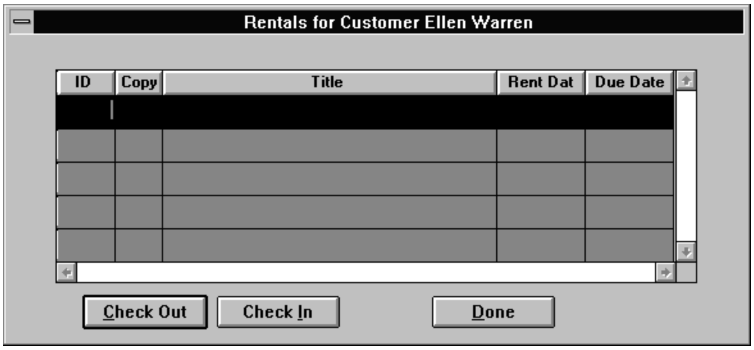


Figure B-7 The Rental Listing screen displays videos currently rented by the customer (indicated in the screen's title bar).

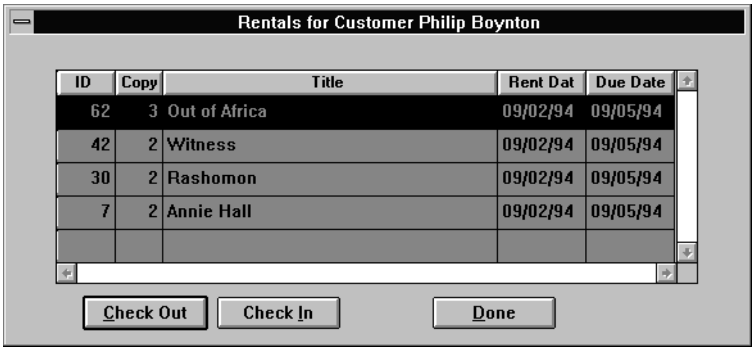
The selected customer’s identification number and name are sent to the Video Rentals (`rentlist.jam`) screen; the customer's name is displayed in the screen’s title bar. All videos that are currently rented by the selected customer are listed; if there are no videos out, the grid is empty, and ready for Check out.

From the `rentlist` screen, the user can:

- Choose the Check Out button to rent additional video titles. The Video Rental (`rentvid.jam`) screen is displayed.
- Choose the Check In button to return videos.

How to Return a Video

1. Select the video from the list.


Rentals for Customer Philip Boynton

ID	Copy	Title	Rent Dat	Due Date
62	3	Out of Africa	09/02/94	09/05/94
42	2	Witness	09/02/94	09/05/94
30	2	Rashomon	09/02/94	09/05/94
7	2	Annie Hall	09/02/94	09/05/94

Figure B-8 The rental date and due date are indicated for each video title.

2. Choose the Check In button.

Repeat these steps for each return.

Behind the Scenes

The `rentlist.jam` screen includes the following features:

- The screen receives the `cust_id` and the customer name from the `custlist.jam` screen and runs a query to find what videos the customer currently has out.
- The customer’s name is received into the screen's title bar (this specification is defined in the screen’s Title property under Identity).
- When the Check Out button is armed, it sends the `cust_id` to the New Rentals (`rentvid`) screen and invokes the `rentvid` screen.
- The Check In button updates the rental status and redisplays the screen.

Rent Videos

It is expected that in this video store, the customer brings the video cassette to the front desk when he or she wants to rent a movie. The video itself has the ID and copy number printed on the container. The front desk clerk can just type that information into the application.

How to Rent a Video

1. From the Rentals (`rentlist`) screen, choose the Check Out button.

   The New Rentals (`rentvid.jam`) screen opens.

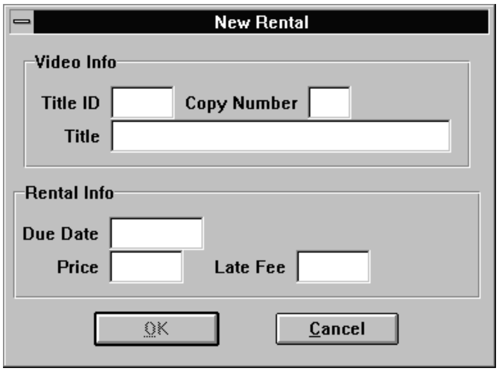
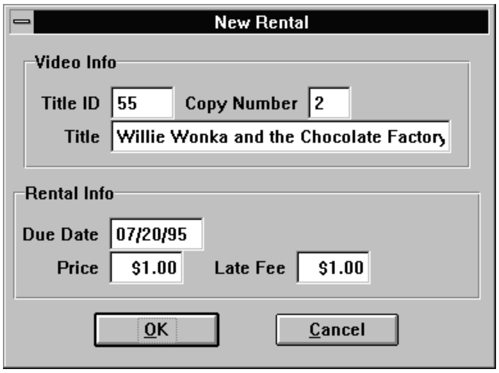


Figure B-9 New rentals require a title ID.

2. Enter a title ID and press TAB.

   The application displays the video title associated with the specified ID number. If a title ID does not exist, an error message is displayed.

3. Enter the tape copy number (usually a number between 1 and 3 inclusively) and press TAB.



Title ID: 55
Copy Number: 2
Title: Willie Wonka and the Chocolate Factory
Rental Info
Due Date: 07/20/95
Price: $1.00
Late Fee: $1.00

Figure B-10 An available copy of the video automatically provides the due date and price of the rental as well as the late fee charge.

If the video associated with the specified ID and copy number is, in fact, available, the cost of the rental, late fee rate, and due date are displayed.

If the specified copy number is not available (it's already rented by another customer or does not exist), an error message is displayed. Another number can be entered.

4. Choose OK to record the rental. The New Rentals screen closes, and the Rentals list is updated with the newly rented video titles.

Behind the Scenes

The `rentvid.jam` screen includes the following features:

- Receives the `cust_id` from the Rentals (`rentlist`) screen.
- After an available video tape is specified, the screen is prepared for an insert to the database. It uses `sm_tm_command("SAVE rentals TV-ONLY")`
- A transaction event function `rental_hook` is invoked after a new rental is inserted in the database. It updates the `tapes` database table by logging the tape as "unavailable."
- Review the screen-level JPL module; most of these procedures apply the business logic required to make this screen work correctly.

Customer Profile

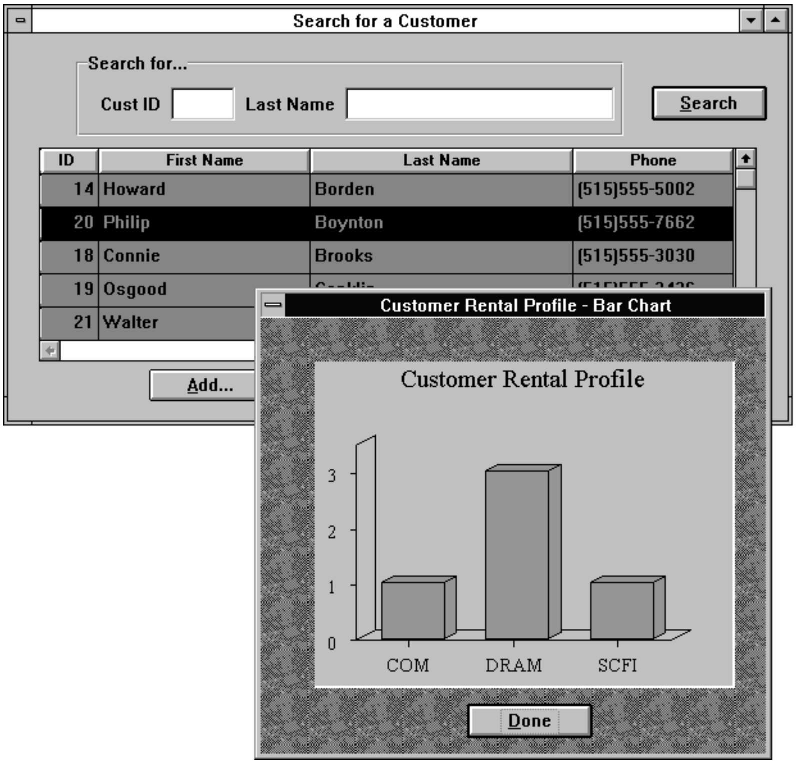
The Customer Profile options are only available to a user logged in as an Employee. A customer profile provides two different graphical representations of the types of videos a selected customer has rented.

How to Obtain a Customer Profile

1. Select a customer from the Search for a Customer (`custlist`) screen.
2. Choose the design chart format by doing either of the following:

   - Choose Options→Profile→Bar Chart or 📊

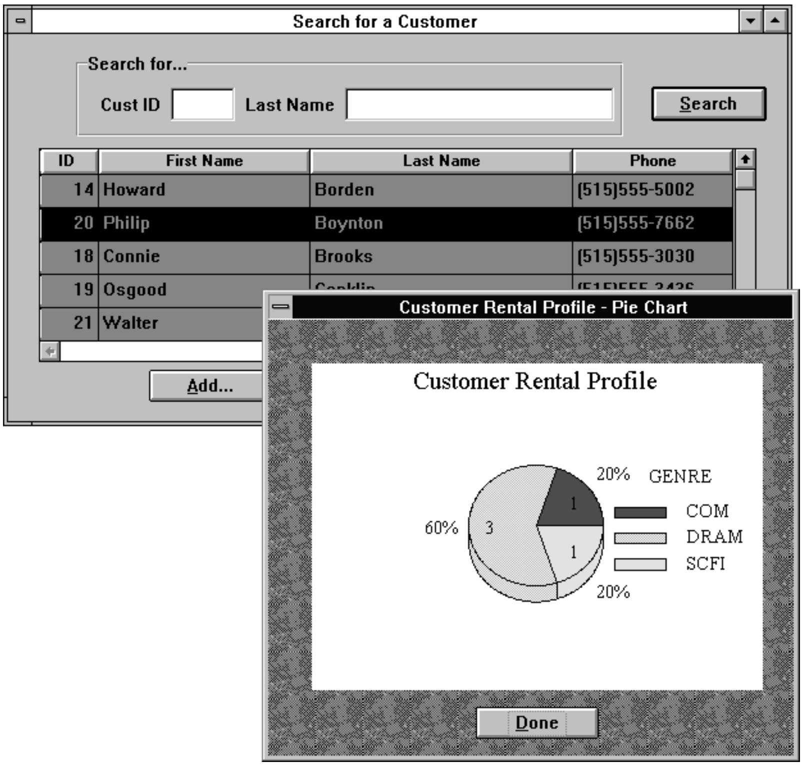
     The selected customer's rental profile is displayed in the bar chart format.



ID	First Name	Last Name	Phone
14	Howard	Borden	(515)555-5002
20	Philip	Boynton	(515)555-7662
18	Connie	Brooks	(515)555-3030
19	Osgood		
21	Walter		

Figure B-11 Rental profile illustrates the customer's rental preferences by category.

- Choose Option→Profile→Pie Chart or



ID	First Name	Last Name	Phone
14	Howard	Borden	(515)555-5002
20	Philip	Boynton	(515)555-7662
18	Connie	Brooks	(515)555-3030
19	Osgood	Conklin	(515)555-3436
21	Walter		

Figure B-12 The selected customer's rental profile is displayed in pie chart format.

3. Choose Done to return to the Search for a Customer screen.

Behind the Scenes

The bar chart and pie chart screens include the following features:

- The `cust_id` is passed from the `custlist` screen and is used to execute the query.
- The count of each video category (drama, comedy, etc.) is gathered from the `rentals` table using the `count(*)` expression. Panther automatically performs a group by category.

Video Lookup

The user can access the video lookup portion of VideoBiz by doing either of the following:

- Logging into VideoBiz as a customer (choosing the Customer radio button on the Welcome screen)
- Choosing Options→Video Search or

The Video Lookup consists of two screens: Video Listing and Video Detail. The Listing screen allows the user to search for a video by title ID, movie title, or director. It fetches all records that match the search criteria and displays the results in scrolling lists. From this screen, the user can access the Video Detail screen, which displays all information about a selected video.

Querying the Database and Selecting a Video

The Search for a Video (`vidlist.jam`) screen allows the user to search for a video and select one from the results. The screen consists of four query fields and a grid widget that contains four columns for displaying query results.

The grid widget lists the video ID, movie title, director, genre, and rating. The grid widget can be shifted from left to right to display offscreen columns. A bounce bar can be moved up and down in the list to indicate the currently selected title. The screen includes a Search button to execute the query, a Detail button that invokes the video detail screen, and a Cancel button to return the user to the calling screen.

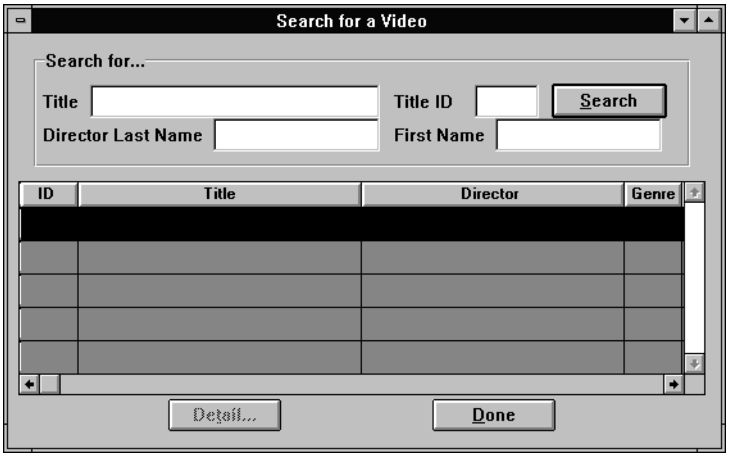


Figure B-13 Query for videos using title, ID, or the director's name.

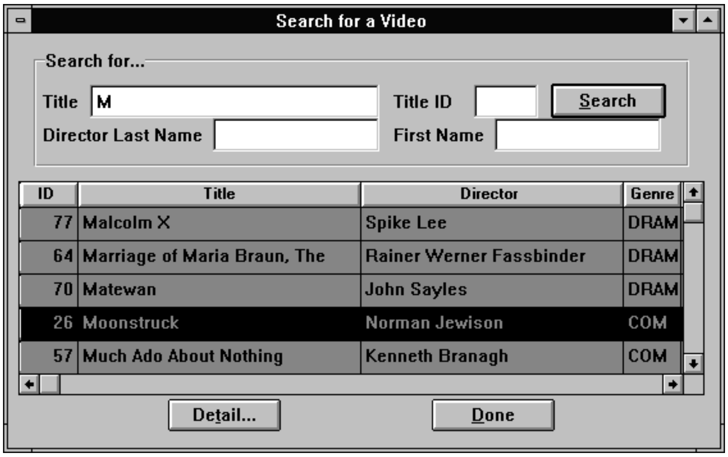
How to Search for a Video

1. Specify the search criteria by entering a title ID or a combination of one or more of the following:

   - Enter a full or partial string in the Title field at the top of the screen, and then choose the Search button or press Enter.
   - Enter a full or partial string in the Director Last Name field and choose the Search button or press Enter.
   - Enter a full or partial string in the Director First Name field and choose the Search button.

   All records that match the search criteria are displayed in the grid widget in alphabetical order by title. In addition to the title and director's name, the rating (e.g., PG (parental guidance), R (restricted), etc.) and genre (e.g., comedy, science fiction) of each title are displayed. The arrays scroll to accommodate more titles than can fit on the screen, and a bounce bar allows the user to select a video title from the list. A horizontal scroll bar allows the user to display offscreen columns in the grid widget.

   If the user chooses the Search button without specifying any search criteria, all videos in the database are displayed in alphabetical order by title.



Title: M

ID	Title	Director	Genre
77	Malcolm X	Spike Lee	DRAM
64	Marriage of Maria Braun, The	Rainer Werner Fassbinder	DRAM
70	Matewan	John Sayles	DRAM
26	Moonstruck	Norman Jewison	COM
57	Much Ado About Nothing	Kenneth Branagh	COM

Figure B-14   Query results are displayed in a grid widget.

2. Select the desired record by doing any of the following:
   - Click on any of the grid members.
   - Use the up and down arrow keys to position the bounce bar on the desired record.
   - Double-click on the desired record. This invokes the Video Detail screen.
3. Choose the Detail button. The Detail button invokes the Video Detail screen.
4. Choose Done to close the Search for a Video screen. If the user is a customer, the Welcome screen appears. Otherwise, the user resumes on the screen that was open when he or she chose the Video Search option.

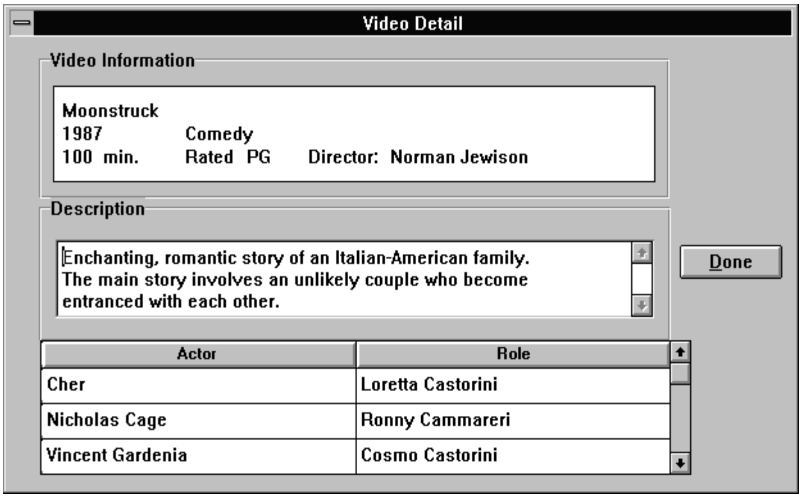
Behind the Scenes

The `vidlist.jam` screen includes the following features:

- Functions in the same way as the customer search screen (`custlist.jam`).
- The `title_id` selected in the grid widget is sent to the video detail screen.

View Video Details

The Video Detail screen (`viddtl.jam`) displays detailed information about a selected video title. The user arrives at this screen as a result of specifying search criteria, querying the database, and selecting a video title from the results. The upper portion of the screen displays general information about the selected video (for example, title, length in minutes, rating code, and pricing category (displays only if the user is an employee)). The middle portion displays a scrolling text area with a description of the video. The grid widget in the lower portion of the screen displays the actors who appear in the film and the roles they play.



Moonstruck
1987 Comedy
100 min. Rated PG Director: Norman Jewison

Enchanting, romantic story of an Italian-American family. The main story involves an unlikely couple who become entranced with each other.

Actor	Role
Cher	Loretta Castorini
Nicholas Cage	Ronny Cammareri
Vincent Gardenia	Cosmo Castorini

Figure B-15 Video details display data from three database tables: titles, title_desc, and actors.

When the user is finished reading the details of the video, choosing the Done push buttons closes the Video Detail screen and returns the user to the Search for a Video screen.

Behind the Scenes

The `viddtl.jam` screen includes the following features that you can examine by accessing the screen editor:

- Open the DB Interactions window to see how the table views on this screen are linked. The screen executes two server joins and two sequential joins.

- A query is executed using `title_id` received from the `vidlist.jam` screen to find the description associated with the selected video title.
- The title ID and its pricing category are displayed conditionally. When the screen opens, the screen entry procedure (`viddtl_se`) checks the `user_type`. If the `user_type` is Customer, the call to the `expose_fields` procedure will hide these field. Otherwise, the fields are displayed to the video store employee.
- The director's first and last name are concatenated into a single field in the procedure defined as this screen's entry procedure (`viddtl_se`). This ensures that when the screen opens, the data are displayed correctly.

Marketing

You can access the Marketing portion of VideoBiz by choosing Options→Marketing from the menu bar or one of the corresponding toolbar items. Data are passed to reports and displayed.

How to Run Marketing Reports

1. The user must log onto the application as an employee to active the marketing menu/toolbar options.
2. Choose the desired report:

   - Choose Option→Marketing→Top Ten or

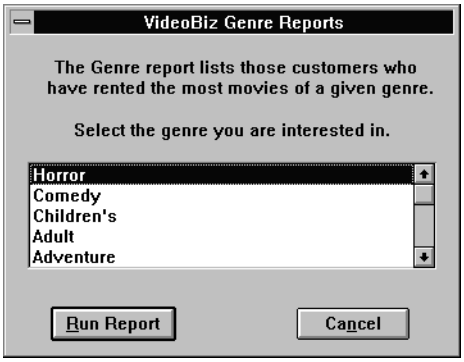
     Lists the ten most frequently rented videos.

   - Choose Option→Marketing→Due Notice or

     Produces a letter addressed to each customer who has overdue rentals as of this reporting period. The letter lists each overdue rental by title and the amount due.

   - Choose Option→Marketing→Genre or

     The user is prompted to select from a list of genre categories.



The Genre report lists those customers who have rented the most movies of a given genre.

Select the genre you are interested in.

Figure B-16 You can generate a report by choosing a particular category of video.

- Select a category and choose Run Report.

  The report lists all those customers who have rented videos having the specified genre. For example, the user can specify the genre Drama. The output lists all customers, in descending order of the number rented, who have rented videos classified as Drama. The customers' phone numbers are listed as well.

APPENDIX

C Panther Java Calculator

This appendix contains the development notes for the Panther Java Calculator sample that is located at `$SMBASE/samples/javacalc`.

Repository Contents

The repository, `calc.dic`, contains the following items.

Table C-1 Lower Buttons

Repository Name	Function	Java Class
`num_pb`	Numeric Keys (also period and +/-)	`CalcNum`
`op_pb`	Arithmetic operations	`CalcFunc2`
`mem_pb`	Memory functions	`CalcMem`
`misc_pb`	Miscellaneous operations (Constants, Clear, backspace)	`CalcMisc`

Table C-2 Upper Buttons

Repository Name	Function	Java Class
`trig_pb`	Trigonometric functions	`CalcTrig`
`mode_pb`	Mode change	`CalcMode`
`func1_pb`	Single operand functions	`CalcFunc1`
`func2_pb`	Two operand functions	`CalcFunc2`

Calculator Screen

The calculator screen, `calc.scr`, contains the following widgets and settings.

Table C-3 Screen Properties

Property	Setting
IDENTITY→Title	Calculator
IDENTITY→Java Tag	CalcScreen
HELP→Help Screen	help.scr
FOCUS->JPL Procedures	`global register, memory, operation, op_just_done, degrees, currency` ■ `register`: Holds the "invisible" operand (last number entered before performing operation) ■ `memory`: storage for M+, MR, MC function ■ `operation`: Storage of the last 2 operand operation key pressed ■ `op_just_done`: Used to determine if next number entered should clear display first. ■ `degrees`: flag for trig operations, 1=degrees, 0=radians ■ `currency`: flag for display, 1=fixed decimal (2) places, 0=floating decimal

Table C-4 Display Widgets

Widget Name	Widget Type	Function	Java Class
mode_display	Dynamic Label	Displays modes degrees / radians, currency / normal	None
display	Single line text	Calculator display	None

The following table lists the types of push buttons, their names, the push button they inherit from in the repository and the java class.

Table C-5 Button Widgets

Type	Name	Inherit From	Java Class
Digits: 0,1,2,3,4,5,6,7,8,9	n*X*_pb (where *X* is the digit)	Num_pb	CalcNum (Inherited)
. (period)	dot_pb	Num_pb	CalcNum (Inherited)
+/-	sign_pb	Sign_pb	CalcFunc1
C, CE	c_pb, ce_pb	misc_pb	CalcMisc (Inherited)
*, /	mult_pb, div_pb	op_pb	CalcFunc2 (Inherited)
+, -	plus_pb, minus_pb	op_pb	CalcFunc2 (Inherited)
= (Equals)	equls_pb	op_pb (size changed)	CalcFunc2 (Inherited)
MC, MR	mc_pb, mr_pb	mem_pb	CalcMem (Inherited)
M+	mp_pb	mem_pb	CalcMem (Inherited)
Pi	pi_pb	misc_pb	CalcMisc (Inherited)
_ (Backspace)	bs_pb	misc_pb	CalcMisc (Inherited)
Dg/Rd	angle_pb	mode_pb	CalcMode (Inherited)
#/$	currency_pb	mode_pb	CalcMode (Inherited)
X^Y, Mod	power_pb, mod_pb	func2_pb	CalcFunc2 (Inherited)
Sin, Cos, Tan, Atan	sin_pb, cos_pb, tan_pb, atan_pb	trig_pb	CalcTrig (Inherited)
1/x (Inverse)	inv_pb	func1_pb	CalcFunc1 (Inherited)
Sqrt (Square root)	sqr_pb	func1_pb	CalcFunc1 (Inherited)
Log, Exp	log_pb, exp_pb	func1_pb	CalcFunc1 (Inherited)

Java Classes

The Java classes have been broken into handlers which handle either screen entry or "classes" of buttons. This is a mid-way point between having a separate class for each button and having a single class handle all the buttons.

`CalcScreen`:

Implements ScreenHandler interface

Screen entry function. Initialize global variables and allow backspace, delete and Newline to be caught by ButtonHandler.

Screen exit function. Release backspace, delete, and Newline, restoring normal behavior for those keys.

`CalcFunc1`:

Implements ButtonHandler interface.

Handles all single operand calculator functions (like Sqrt functions). Also handles the Sign toggle (+/-)

`CalcFunc2`:

Implements ButtonHandler interface

Handles all dual operand calculator functions. This class actually performs the last stored operation, and stores the next operation in the global variable "operation."

`CalcMem`:

Implements ButtonHandler interface

Handles all memory functions.

`CalcMisc`:

Implements ButtonHandler interface

Handles C, CE, backspace and Pi button.

`CalcMode`:

Implements ButtonHandler interface

Handle mode switching buttons, "Deg/Rad" and "#/$"

`CalcTrig`:

Implements ButtonHandler interface.

Even though these functions are similar to the CalcFunc1 operations, these make use of the "degrees" setting, so a separate class was created.

`CalcNum`:

Implements ButtonHandler interface.

Handles all numeric keys, and period.

APPENDIX

D Deployment Checklist for Two-tier Applications

Directory Structure for Two-tier Applications

Distribute the files and libraries used by your Panther application in a single directory, call it the application directory. The directory should include such things as your application's executables and Panther-specific libraries. In addition, it should include the following subdirectories:

- configuration directory—Includes the runtime components that make up your application, such as your application libraries and those files that are specific to running your application.
- library directory for UNIX only—Includes JetNet shared libraries. Required for three-tier processing.

Checklist for Deployment

The tables in this section list the components you should include in a distribution for the specific platform. Depending on your particular application, there might be other considerations and files which you might include. Those considerations are covered later in this chapter.

Preparing a Windows Distribution

Table D-1 lists the files and libraries required on a Windows installation. The table also includes where these files can be found in the Panther distribution. In general, you should make copies of those files as opposed to using the originals. In all likelihood, your Panther application has been using the components it needs while you've been developing it. This list will serve as a means of making certain all the pieces you need are deployed to the application users.

Table D-1 Checklist for contents of a Panther application for Windows

File/Library	Found in Panther	Description
application directory contents:		
`cktbl32/64.dll`	`util`	Panther-specific DLL
database DLLs	`util`	Support Panther database drivers—Informix, ODBC, Oracle, Sybase
`*.ini`	`config`	Initialization file `prol5w32/64.ini`
`libsti32/64.dll`	`util`	Panther-specific DLL
`libsti.ini`	`config`	Graph-specific initialization file (copy this file to the Windows directory)
`libxml2.dll`	`util`	Needed if XML files are to be imported.

Table D-1 Checklist for contents of a Panther application for Windows

File/Library	Found in Panther	Description
`msvcr80.dll`	`util`	Microsoft Visual C++ 2005 runtime DLL. Needed if the Redistributable Package will not be installed.
`PanPDF32/64.dll`	`util`	Needed if PDF reports will be created.
`projpeg.dll`	`util`	Needed to process JPEG images.
`promfc32/64.dll`	`util`	Contains status line and frameset code.
`prores32/64.dll`	`util`	Panther Windows resource DLL
`prorun32/64.exe`	`util`	Runtime executable (rename for your application)
`rwres32/64.dll`	`util`	Report Writer Windows resource DLL
config directory contents:		
`client.lib` includes:		
client screens		Panther screens that make up the user interface
`smwzmenu`		Binary menu script file; include if client screens created with screen wizard use the prototype menu bar/toolbar
`smwizard.bin`		JPL module made public by client screens created by the screen wizard
JPL modules		JPL code used by client screens
Graphics files		Image files (such as `*.ico, *.bmp, *.jpg)` referenced on client screens and/or toolbars
`styles.sty`		Transaction manager styles file for your application
`wincmap.bin`	`config`	Binary configuration map file (maps Panther fonts to Windows-specific fonts, etc.).
`winkeys.bin`	`config`	Binary key files for mapping physical keys to Panther logical keys. Omit this file from the library if end-users can modify key mapping on installation.
`msgfile.bin`	`config`	Contains messages and information used by Panther

Table D-1 Checklist for contents of a Panther application for Windows

File/Library	Found in Panther	Description
`*.fnt`	`config`	Graph-specific fonts referenced in graphs in your application
`grafcap`	`config`	Initialization file for graph support
`prorun5.lib`	`config`	Panther's runtime support library
`prorw5.lib`	`config`	Panther's runtime library for reports
`winkeys`	`config`	ASCII key file for mapping physical keys to Panther logical keys. Required if key mapping is user configurable; include `key2bin` utility as well.
`smvars.bin`	`config`	Binary environment setup file

Preparing a UNIX Distribution

Table D-2 Checklist for contents of a Panther application for UNIX/Motif platforms

File/Library	Found in Panther	Description
application directory contents:		
Prolifics	`config`	Resource file for Motif (installation should copy Prolifics to the home directory of each user)
`prorun`	`util`	Client executable (rename for your application); required only if supporting UNIX clients
config directory contents:		
`client.lib` includes:		Required only if supporting UNIX clients
client screens		Panther screens that make up user interface
`smwzmenu`		Binary menu script file; include if client screens created with screen wizard use the prototype menu bar/toolbar
`smwizard.bin`		JPL module made public by client screens created by the screen wizard

Table D-2 Checklist for contents of a Panther application for UNIX/Motif platforms

File/Library	Found in Panther	Description
JPL files		JPL files used by client screens
Graphics files		Image files (e.g., *.xbm, *.xpm, *.bmp, *.jpg) referenced on client screens and/or toolbars
`styles.sty`		Transaction manager styles file
`*cmap.bin`	`config`	Binary configuration map file (maps Panther fonts to Motif-specific fonts, etc.).
`*key.bin`	`config`	Binary key files for mapping physical keys to Panther logical keys. Omit this file from the library if end-users can modify key mapping on installation.
`msgfile.bin`	`config`	Contains messages and information used by Panther
`*vid.bin`	`config`	For character-mode only. Binary files that describe terminal capabilities and attributes to Panther. Omit this file from the library if end-users can modify video specifications on installation.
`*.fnt`	`config`	Graph-specific fonts referenced in graphs in your application; required if supporting UNIX clients
`gdsp`	`util`	Graph support utility
`grafcap`	`config`	Initialization file for graph support
`prorun5.lib`	`config`	Panther's runtime support library
`prorw5.lib`	`config`	Panther's runtime library for reports
`smvars.bin`	`config`	Binary environment setup file. Copy and modify for your application.
`swsdrvr`	`util`	Graph support utility

Index