JAM 7

Application Development Guide

August 1995

This software manual is documentation for JAM® 7. It is as accurate as possible at this time; however, both this manual and JAM itself are subject to revision.

Send suggestions and comments regarding this document to:
Technical Publications Manager
JYACC, Inc.
116 John Street
New York, NY 10038
(212) 267–7722

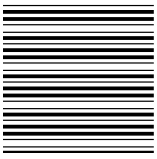
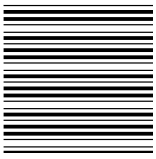


Table of Contents



About this Guide

This guide covers various topics related to application development. It discusses approaches to development, strategies for using JAM effectively, and the order in which tasks should be performed.

You should read Chapter 1 for a comprehensive overview of JAM as a development tool. If you want a detailed example of an application developed with JAM, look at Chapter 2, which describes the Videobiz sample application that is part of the JAM distribution. The rest of the guide is topical, and should be used as a reference when specific application development issues arise during your development cycle.

Note: *JAM's proprietary programming language, JPL is fully discussed in the Language Reference.*

Organization of this Guide

This guide is organized into six sections:

Section One: Overview
A comprehensive overview of the development process and detailed description of the Videobiz sample application.

Section Two: Application Building Blocks
Information about the various components in a JAM application, including screens, widgets, repositories, menu bars, and hook functions.

Section Three: The SQL Executor
The protocol for JAM's interaction with your database engine—how to initialize and connect to database engines as well as how data and status information is fetched from, or written to, the database.

Section Four: SQL Generation
How JAM's SQL generator constructs SQL statements from widget, link, and table view properties.

Section Five: The Transaction Manager
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 Six: Application Issues
Topics related to JAM's runtime processing, portability, internationalization, packaging, and debugging.

Conventions

The following typographical and terminological conventions are used in this guide:

Text Conventions

`expression` Monospace (fixed-spaced) text is used to indicate:

- Code examples.
- Words you're instructed to type exactly as indicated.
- Filenames, directories, library functions, and utilities.
- Error and status messages.

`KEYWORDS` Uppercase, fixed-space font is used to indicate:

- SQL keywords.
- Mnemonics or constants as they appear in JAM include files.

numeric_value Italicized helvetica is used to indicate placeholders for information you supply.

[option_list]	Items inside square brackets are optional.
`{x | y}`	One of the items listed inside curly brackets needs to be selected.
x ...	Ellipses indicate that you can specify one or more items, or that an element can be repeated.
new terms	Italicized text is used: ❍ To indicate defined terms when used for the first time in the guide. ❍ Occasionally for emphasis.

Keyboard Conventions

XMIT	JAM logical keys are indicated with uppercase characters.
Alt+A	Physical keys are indicated with initial capitalization, and keys that you press simultaneously are connected with a plus sign.

JAM Documentation

The JAM documentation set includes the following guides and reference material:

Read Me First — Consists of three sections:

- *What's New in JAM* — Briefly describes what's new in JAM 7.
- *Installation Guide* — Describes how to install JAM on your specific platform and environment.
- *License Manager Installation* — Instructions for installing the License Manager (used on many UNIX and VMS platforms).

Getting Started — Contains useful information for orienting you to JAM. Includes a description of the JAM environment and features, how JAM addresses real-world application development issues, and a guided tutorial for building a mini-JAM database application.

Editors Guide — Instructions about using the JAM authoring environment; learn how to use the graphical tools for creating, editing, and designing your application interface. Includes detailed descriptions of the screen editor, screen wizard, menu bar editor, and styles editor. The *Editors Guide* is also provided online on GUI platforms. It is installed with the installation of the JAM software and can be accessed by selecting help from within the screen editor.

Application Development Guide — Information by topic to guide you in developing your JAM application. This includes components of the JAM development environment such as the repository, hook functions, and menu bars, as well as sections on the SQL executor, SQL generator and the transaction manager.

Language Reference — Describes JPL, JAM's proprietary programming language. Also includes reference sections for JPL commands, built-in functions and JAM library functions. The man pages in the reference sections are arranged alphabetically.

Database Guide — Instructions for using JDB, JYACC's prototyping database, and for the commands and variables available in the database interfaces. Includes an Database Drivers section containing instructions unique to each database driver.

Configuration Guide — Instructions for configuring JAM on various platforms and to your preferences. Some options that can be set relate to messages, colors, keys and input/output. Also includes information on GUI resource and initialization files.

Master Index and *Glossary* — Provides an index into the entire documentation set and a dictionary of terms used in the documentation set. This is in addition to the indexes in the individual volumes.

Upgrade Guide — Online only. Information for upgrading from JAM 5.

Online Documentation

JAM's documentation set is available online and included with the JAM distribution. The books can be viewed through the DynaText™ browser on GUI platforms. It can be accessed by choosing Help from within JAM or by running DynaText's read-only browser from the command line or by clicking on the DynaText icon. For instructions on using DynaText, request Help while you have a browser window open.

Collateral Documentation

The following information is also provided with your JAM installation:

- Database Driver Notes — JAM 7 has database drivers for most popular relational database engines, as well as JDB, JAM's proprietary database. Information for JDB, Sybase, Oracle, Informix and ODBC are located in the *Database Guide*; others are included separately.
- Online help — The *Editors Guide* is provided in online form through the DynaText browser on GUI platforms. It can be accessed by choosing Help from the screen editor. For instructions on using DynaText, request Help while you have a browser window open.
- Online README file.

Additional Help

JYACC provides the following product support services; contact JYACC for more information.

- Technical Support
- Consulting Services
- Educational Services

SECTION ONE

Overview

JAM Development Overview

JAM helps developers to prototype and build powerful and efficient applications that can operate in a variety of computing environments. JAM has been used to build a broad range of applications, such as order entry, customer service, project planning, transportation logistics, and manufacturing automation.

There is no single correct approach to building a JAM application. In fact, JAM provides you—the professional developer—with tools that support many different approaches. The goal of this development overview is to demonstrate one approach while providing sufficient detail to let you perceive other approaches.

We'll demonstrate the approach using a small portion of a video store application as an example. The next chapter in this guide (page 33) fully documents a video store application that is distributed as a sample application with JAM. In this chapter, our application will display each of the actors and their roles for the specified movie title. For this purpose, we'll create the movie screen shown in Figure 1.

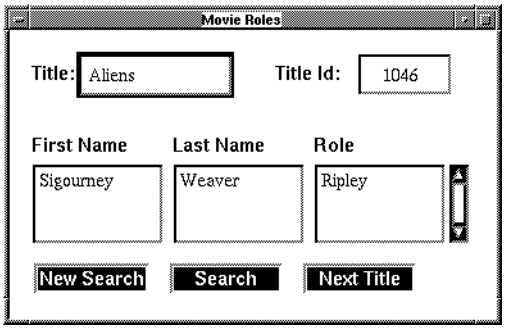


Figure 1. Sample movie screen.

What You Need to Use JAM

JAM supports many graphical user interfaces (GUIs) and databases. To develop and run JAM applications with a specific GUI and database, you need:

- A presentation interface for the GUI.
- A database driver for the database.

JAM ships with at least one presentation interface, along with a database driver for JDB, JAM's built-in database for prototyping.

Components of a JAM Application

A JAM application typically comprises:

- Screens
- Menus and toolbars
- JPL modules
- Configuration files
- Application executable
- Database

There are several ways to package a JAM application for distribution. Typically, the screens, menus, and JPL are shipped in one or more libraries. The configuration files and the application executable are shipped as separate files. If desired, the screens, menus, JPL modules, and the configuration files can be shipped as part of the application executable. Packaging is described in more detail later in this chapter and in Chapter 29.

Components of the JAM Authoring Environment

Authoring is the process of developing a JAM application. A JAM authoring environment typically comprises the screens, menus, toolbars, JPL modules, configuration files, and database that will become part of the application, plus:

- Screen editor
- Menubar editor
- Styles editor
- JAM Debugger
- Repository
- Authoring executable (`jamdev`)
- Configuration manager. JAM works with third-party configuration management tools such as SCCS and PVCS. Customers may elect to build their own interfaces to these or other configuration management tools. Configuration management currently applies to JAM screens.

The JAM authoring environment optionally includes other add-on products: JAM/CASE interface, JAM/TP*i*, and JAM/ReportWriter.

Creating Screens

To create screens, you must start the authoring executable, `jamdev` which invokes the screen editor. The screen editor includes a menu bar, a toolbar, a tool box, a color palette, a widget list, a property window, and one or more drawing areas. Each drawing area contains a separate screen. Typically, the screen editor starts with a single, empty, drawing area. The screen can be enhanced by using tools selected from the toolbox (or from the menu bar). Since the screen editor can open many screens at once, existing objects can be copied between screens via drag and drop. The screen editor is fully documented in the *Editors Guide*.

Figure 2. The JAM screen editor

To maximize development productivity by promoting object reuse, and to reduce maintenance costs, you may wish to create screens by copying screen objects from JAM's *visual object repository,* which is described later.

Iterative Application Development

Application development is never done with perfect knowledge of the true requirements. Often requirements change based upon the knowledge gained as an application is developed. Therefore, JAM fully supports an iterative and interactive approach to application development. Changes can be made and tested without leaving the authoring executable, resulting in rapid feedback to both developers and users. In fact, changes can be made while the application is being demonstrated to the user.

Iterative development works because from within `jamdev` you have immediate access to:

- Application mode — In this mode, you can test the entire application as though you are a user. The advantage of using application mode within `jamdev`, rather than using the application executable itself, is that you can enter the screen editor to modify whatever screen is currently displayed.
- The screen editor — In the screen editor you can make changes to your screens. From the screen editor you can either return to application mode to fully test the entire application, or you can enter test mode.
- Test mode — Test mode is entered from the screen editor. It is similar to entering application mode, starting at the screen that is currently being edited. It allows the developer to immediately see and use the screen, without needing

to permanently save that screen and without losing the state of the screen editor. In test mode you can open other screens as well. When test mode is exited, however, the current screen is not opened for editing, but rather the editor opens in the state it was in before test mode was entered.

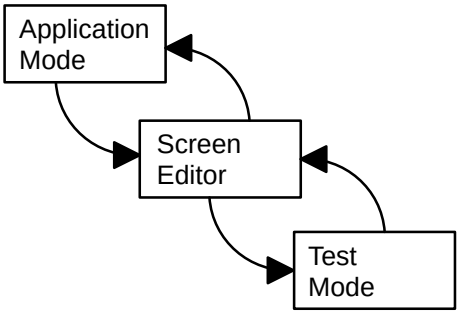


Figure 3. Iterative application development.

The Visual Object Repository

The repository is the key to sharing and reusing objects within a JAM application. The repository is a JAM library that consists of one or more members. Each member is, in fact, a JAM screen. Therefore, the repository screens can be viewed and changed with the screen editor. The advantage of keeping repository objects in JAM screens is that it makes the repository *visual*. Each object can be viewed as it will appear, rather than as part of a list. In addition, each object can be viewed in relation to other objects with which it is normally associated. Consider the simple example of a person's name. On screens throughout an application, you might expect it to look something like:

```
First: __________   Last: ________________
```

In a non-visual repository the name might be stored in as many as four objects, one per field and one per field label. Copying each object individually to a new screen to recreate the name would be tedious and error-prone. Since JAM's visual object repository preserves spatial relationships, a widget and its label can be copied as a single drag and drop operation as shown in Figure 4.

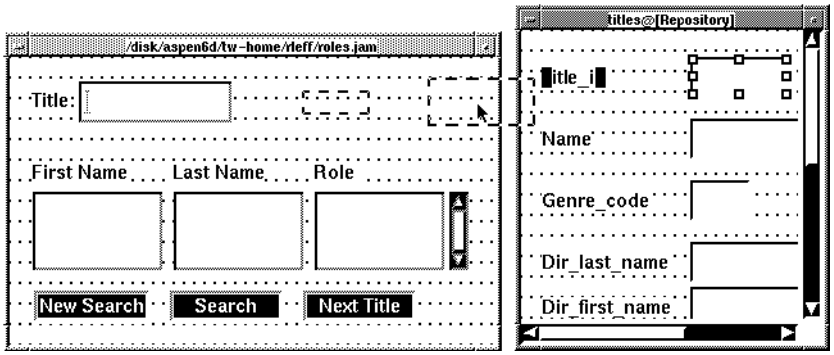


Figure 4. Copying the title_id label and widget from the visual object repository (on the right) to the movie screen (on the left).

The Repository and Inheritance

By copying an object, such as a text widget, from the repository to a screen, you create an inheritance relationship between the repository object and the screen object. If you subsequently change a property of the repository object, then the change can be propagated to all screen objects that inherit from the repository object. The propagation of inheritance, called inheritance resolution, happens automatically within the screen editor and as requested via the batch inherit utility, `binherit`. Inheritance can also be established between objects within the repository, but not between objects within screens. Inheritance can be overridden on a property-by-property basis.

The inheritance relationship between an object and the repository is preserved, even when the object is copied between non-repository screens. The relationship is between the newly-created object on the screen and the repository object. This enables rapid use of existing objects without the need for going back to the repository.

At the present time, inheritance applies to most, but not all, JAM objects. It does not apply to selection groups or menus. Refer to page 61 for details on the repository.

JDB: JAM’s Built-in Database for Prototyping

JDB is a single-user relational database provided with JAM. It provides a SQL interface for data definition and maintenance that is similar to most multi-user

relational databases. The examples in this overview are based upon JDB. Please note that, although the data definition facility includes declarative referential integrity, it is not enforced by JDB. It is present in the data definition language in order to facilitate prototyping of applications. Specifically, JAM can use the information to build SQL statements dynamically. Therefore, you should define referential integrity constraints when using JDB. JDB is described in the JDB section of the *Database Guide*.

Application Development Steps

A JAM application is typically built in an iterative fashion, but it is useful to think of the process as consisting of the following steps:

1. Build the repository from the database.
2. Enhance the repository screens.
3. Create application screens.
4. Customize the application screens.
5. Package the application for distribution.

Creating and customizing an application screen can be further subdivided into the following steps:

1. Copy objects onto the screen from the repository *or* use the screen wizard to create the initial screen.
2. Customize and add screen objects.
3. Define user actions with menu bars, toolbars, push buttons, and function keys.
4. Define additional actions with hook functions.
5. Define and customize the database access. You can use the SQL executor to execute the SQL statements that you write for the screen, or you can use the transaction manager to automatically generate SQL from property values. Both enable the application to access and modify the database.
6. Write specialized logic in JPL.

The following sections describe each of these steps.

Build the Repository from the Database

Although JAM, through prototyping screens, can be used to help develop a database, we'll start developing our application with its database in place. We will

use just three of the tables from the video store application, and show only a few columns for each table. The details of each table are shown below:

```
CREATE TABLE actors (
       actor_id        INT             NOT NULL,
       last_name       CHAR    ( 25)   NOT NULL,
       first_name      CHAR    ( 20)           ,
       PRIMARY KEY (actor_id));

CREATE TABLE titles (
       title_id        INT             NOT NULL,
       name            CHAR    ( 60)   NOT NULL,
       .
       .
       .
       PRIMARY KEY (title_id));

CREATE TABLE roles (
       title_id        INT             NOT NULL,
       actor_id        INT             NOT NULL,
       role            CHAR    ( 40)           ,
       PRIMARY KEY (title_id, actor_id),
       FOREIGN KEY (title_id) REFERENCES titles (title_id),
       FOREIGN KEY (actor_id) REFERENCES actors (actor_id));
```

To build the repository, simply import the database tables into the repository. This involves entering the screen editor, opening or creating a repository, opening the database, choosing File⇒Import from the menu bar, and then selecting the tables to be imported. The import process creates one screen per imported table. In this case, JAM creates three repository screens as shown in Figure 5, one for each selected table.

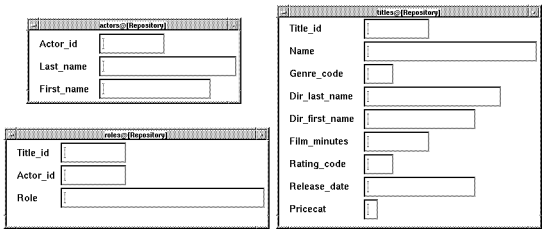


Figure 5. Repository screens created from database tables.

Each screen contains one text widget and one label widget per database column. JAM assigns the database column name to the name of the text widget. Other properties are set similarly to reflect the properties of the column in the database. A table may be reimported in order to update the information kept in the repository. This means that, if you build screens by copying objects from the repository,

changes to database column properties can be propagated throughout your application via inheritance relationships.

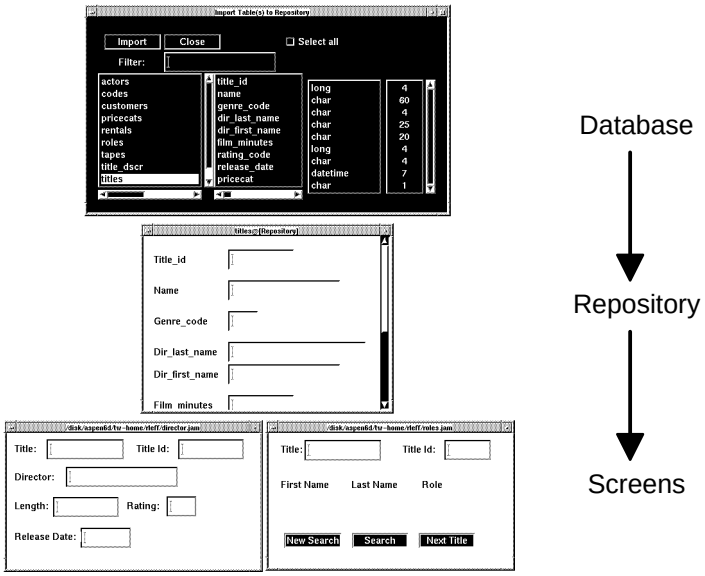


Figure 6. Inheritance from the database to the repository to screens.

Table Views and Links

In order to support automatic database access, JAM needs to know which widgets are associated with which tables, and how the tables are related. Table information is stored in a *table view* widget. 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 must create the links manually. Links and table views are discussed in greater detail in the SQL Generator section of this chapter, as well as in Chapters 20 and 21.

The objects created for each table are listed in the following tables.

Table 1. The actors table — objects created by import.

Object Name	Object Type	Comment
`actor_id`	text widget	
`last_name`	text widget	
`first_name`	text widget	
`actors`	table view	Includes `actor_id`, `last_name` and `first_name`. `actor_id` is marked as primary key.

Table 2. The titles table — objects created by import.

Object Name	Object Type	Comment
`title_id`	text widget	
`name`	text widget	
`titles`	table view	Includes `title_id` and `name`. `title_id` is marked as primary key.

Table 3. The roles table — objects created by import.

Object Name	Object Type	Comment
`title_id`	text widget	
`actor_id`	text widget	
`role`	text widget	
`roles`	table view	Includes `title_id`, `actor_id` and `role`. `title_id` and `actor_id` are marked as primary key.
`K1roles`	link	Links roles and titles tables based on `title_id`. Shown as `roles+titles` on the screen.
`K2roles`	link	Links roles and actors tables based on `actor_id`. Shown as `roles+actors` on the screen.

The import process creates its own link names. You can give these links names that make sense to you. For example, you might change `K1roles` to `link_roles_to_titles`.

Enhance the Repository Screens

JAM's import process is limited by the information available in the database. For example, there is no information about the expected spatial relationship between columns when presented on the screen. There is also no information about the most appropriate widget type. Therefore, you can enhance the repository screens without breaking the ability to reimport tables. Your enhancements might include:

- Rearranging the widgets on the screen.
- Changing the widget size.
- Changing the labels.
- Adding new widgets. For example, you may want to add a total price widget to hold the product of the unit price and the quantity.

imported label widgets get named automatically

JAM assigns names to label widgets that are created by the import process. The name of a label widget is "L" followed by the name of the associated text widget. Since inheritance relationships are based on names, labels on screens will be updated when labels are changed in the repository.

Create Application Screens

using the screen wizard

The easiest way to create new screens is by using the screen wizard. This option is available each time you create a new screen in the editor.

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, item selection screens for any additional tables, and a JPL module containing the JPL procedures needed to execute the transaction manager commands.

copying objects from the repository

Alternatively, you can create new screens by copying objects to the screen either via drag and drop, or copy and paste. To enable propagation of properties via inheritance, it is best if the copied objects either reside in the repository or are set to inherit their properties from objects in the repository. For the example screen, the following diagram depicts the repository source of each of the screen objects that contains data. Refer to page 61 for details on the repository.

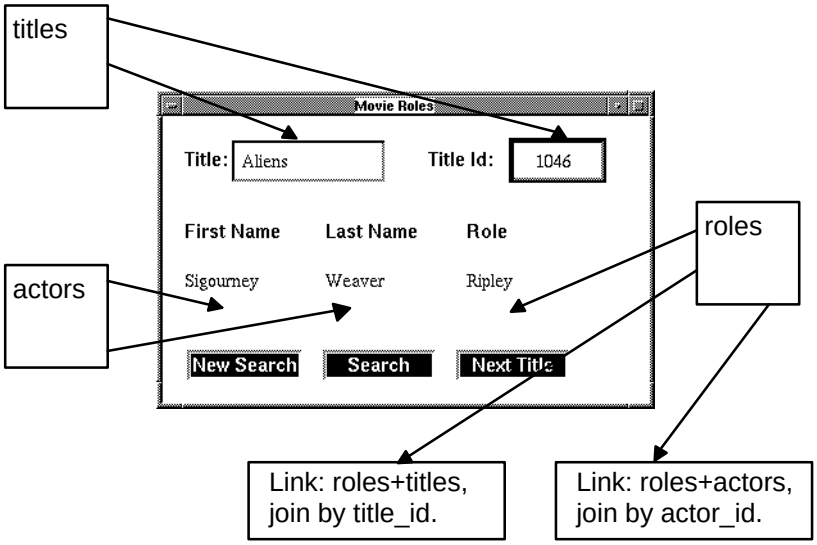


Figure 7. Source of each object on the movie screen.

Customize and Add Screen Objects

You can modify the properties of objects created by the screen wizard or copied from the repository, and add new objects. Many changes typically are directed towards properly using the SQL generator and the transaction manager. For example, you might want to specify a sort order for the list of actors and roles that changes one of the table view properties.

Other changes can customize the presentation of the screen. For example, you might want to change the colors or fonts of various objects or add decorative elements such as boxes or lines. Information can be displayed using different types of business graphics, such as graphs or pie charts.

Define User Actions with Menu Bars, Push Buttons, and Function Keys

Menu items, push buttons, and function keys can be associated with processing logic. In all three cases, this association can be defined with JAM *control strings*. Some sample control strings are shown in the following table. Although it is often

a good idea to keep model push buttons in the repository, we'll create them on the screen for ease of demonstration of our small application.

Control String	Action
`&&detail`	Display the `detail` screen as a sibling (non-modal) window.
`^sm_emsg("Update failed.")`	Call a JAM library function to display an error message. You can also call your own JPL or C functions.
`!date`	Execute the operating system date command.

The control string is a property of push buttons and menu items. Actions for function keys are specified as a property of the screen itself. In addition, function key actions can be defined globally throughout the application.

keep push button templates in the repository

It is a good idea to place commonly used push buttons in the repository. These push buttons can then be copied onto screens as needed.

The actions associated with push buttons on the movie screen are shown in Table 4.

Table 4. Push button actions on the movie screen.

Button Text	Action (control string)	Description
New Search	`^sm_tm_command("CLOSE")`	Discard the current transaction to enable entry of new search criteria.
Search	`^sm_tm_command("VIEW")`	Select title information for viewing purposes only.
Next Title	`^sm_tm_command("CONTINUE")`	Fetch information about the next title.

For detailed information about control string, refer to page 109.

Define Event-Based Actions with Hook Functions

You can specify JPL or C functions to be invoked when certain events occur. In addition, you can specify processing that is to occur for a specific object, such as a single screen, or application-wide for all objects. When processing is specified for

a specific object, 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 the focus, specify `myfunc` as the value of the Entry Function property of the widget (under the Focus heading). When processing is application-wide, the function must be written in C and installed using the library function `sm_install`. For details on hook functions, refer to page 115.

Define Database Access

The transaction manager enables JAM to interact with the database according to user actions. The JAM repository should contain most of the information needed for transaction management, either due to the database import or due to enhancements made to the repository. By copying objects from the repository (or from other screens built from the repository), you minimize the amount of work required to set up the transaction manager. The transaction manager is described in more detail later in this chapter and in Chapters 20 and 21.

Alternatively, the SQL executor 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 SQL executor to process.

Write Specialized Logic in JPL

Much of an application's behavior can be defined by setting object properties rather than by writing code. Where code is required, it can be written in the JPL language, or in C. JPL is usually preferred for this reasons:

- No separate compiling or linking is generally required since JPL is interpreted.
- JPL can be associated with specific screens or widgets.
- JPL has better constructs for handling strings.

Programming in C might be preferred for performance reasons when doing operations iteratively or when doing intensive calculations. It might also be preferred for large programming efforts, such as writing a custom transaction model or other code that will be used repeatedly throughout an application. For details on JAM's C library and JPL, refer to the *Language Reference*.

Create Menus and Toolbars

You can create menus and toolbars in the menubar editor. You can also create menus directly via text scripts. A menu can be associated with a screen either

programmatically, or by setting the properties of the screen. Note that menus can also be associated with the application as a whole and with individual widgets.

For more information about menus, refer to page 87.

Package the Application for Distribution

JAM provides you with many packaging options. Some options improve application performance, others reduce memory use. The options are summarized in Table 5.

Table 5. Application objects and their possible locations in a distribution.

Application Object	File	Library	Screen	Executable
screen	✓	✓ *		✓
JPL	✓	✓ *	✓ *	✓
menu bar	✓ *	✓ *		✓
executable	✓ *			
configuration files	✓ *			✓

✓ * indicates one reasonable approach to packaging.

Two locations are mentioned as reasonable for JPL in Table 5. JPL that is stored in a screen is localized since it can be called only from within that screen. This localization is not always desirable. Two locations are also indicated for menu bars; although menu bars run faster out of libraries, the menu bar editor does not currently let you directly edit menu bars in libraries.

There is no single preferred choice of locations. For example, you might prefer to distribute your application in a form that is most easily maintained via the JAM authoring tools. The screen editor can directly edit screens stored in files and libraries. It can also edit JPL stored in screens. By storing screens in libraries during development, you can let JAM manage concurrent access to libraries. At runtime, libraries can provide better performance than files because JAM maintains an in-memory index of the list of available objects stored in the open libraries, but at the expense of using more memory.

The fastest performance is obtained by linking objects into the executable, at the cost of using more memory. However, it is more difficult to build and maintain all

application objects in a single executable than it is to build and maintain objects as libraries and files.

Adding menu bars and JPL to a library requires use of JAM's library maintenance utility, `formlib`. Adding objects to the executable requires use of JAM's `bin2c` utility. For more information about packaging, refer to page 519.

Levels of Database Access

JAM provides you with four levels of database access:

1. Transaction Manager. The transaction manager determines what SQL must be generated and executed, and asks the next level to do the work.

2. SQL Generator. The SQL generator constructs SQL statements and asks the next level to execute them.

3. SQL Executor. The SQL executor passes SQL requests to the database API, and returns formatted results to JAM. The SQL executor is implemented via the `DBMS` verb in the JPL language, and the JAM library function `dm_dbms`.

4. Database API. This is the API provided by the database vendor.

Although access via the transaction manager is usually easiest, you may use any combination of the levels in your application. For example, you might allow the transaction manager to handle most access itself, but supply specific SQL statements—for example, stored procedure or RPC calls—in certain performance-critical areas. This is easily done with transaction manager hook functions.

The levels of database access are depicted in Figure 8.

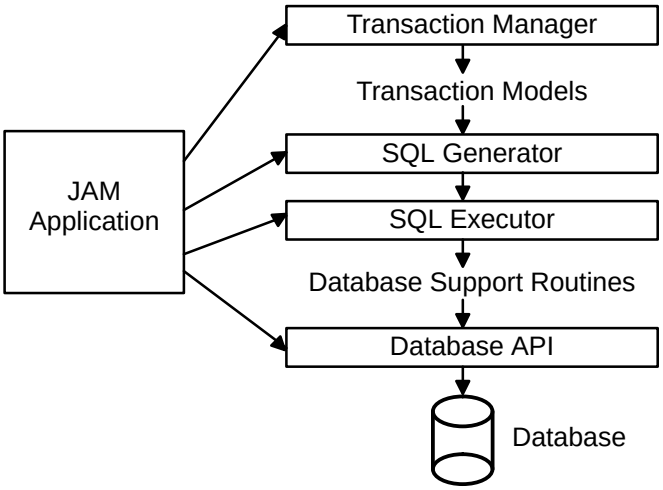


Figure 8. Levels of database access in JAM.

Each level of database access is described below, starting at the lowest level.

Database API

There are two mechanisms for using the database vendor's API. Both require writing C code. The first is to use the API without using any of JAM's other database access facilities. The second, which is currently available for Sybase, is to use the source code provided in the file `usrhndlr.c` to gain access to the database's internal data structures. Because of the complexity of using database APIs, most developers will not use this access level.

SQL Executor

The SQL executor provides an easy-to-use and powerful interface for the execution of SQL statements and other database directives. You create any SQL statement using the SQL dialect supported by your database. This means that you can utilize the most powerful, although possibly non-portable, features of your database such as stored procedure and RPC calls. You can either hard-code these SQL statements, or generate them programmatically with JPL or C. The SQL executor uses the database API to execute the statements and to return result rows, if any.

It is very common to explicitly access the SQL executor while also using the transaction manager. Consider the case of a database that supports stored procedures. The SQL generator does not automatically generate calls to stored

procedures. To replace an automatically generated SQL `INSERT` statement with a call to a stored procedure, you would write a transaction hook function to handle the insert. Your function would call the stored procedure using the SQL executor.

for database portability, use higher level access

For better database portability with respect to SQL, the higher levels of access should be used where possible. On the other hand, you can achieve a reasonable level of portability by using SQL constructs that are common across the relevant databases. However, it is very important to understand how the SQL executor works, since it is used by the SQL generator and the transaction manager.

The SQL executor is very portable with respect to the handling of results sets from the database. This is true for several reasons:

- The SQL executor inspects each column returned from the database to determine the column's data type. It then formats the column's data according to the properties of the destination object. You do not need to use database-specific formatting statements—for example, to format a date column—when writing SQL.
- SQL is a set-oriented language. There are no SQL commands to control fetching of individual rows from a result set. JAM provides fetching commands and supports them across all databases. JAM even supports re-fetching of rows when the database itself offers no such support.

In addition, JAM provides certain other commands that are supported across all databases, such as those for transaction and cursor control.

The JPL interface to the SQL executor is the `DBMS` verb. To execute a SQL `SELECT` statement from JPL, you could use:

```
DBMS SQL SELECT title_id, name FROM titles WHERE name= :+name
```

The SQL executor takes the following actions:

1. The value stored in the `name` widget replaces `:+name`. The proper quotation marks, if needed, are provided automatically since `:+` requests "database smart" substitution via colon plus processing.
2. The `SELECT` statement is sent to the database via the database's API.
3. After the database completes the `SELECT`, JAM obtains the names and types of the columns selected. Since JAM does not parse the select list, it is possible to use any legal database expression in the select list, such as * and aggregation functions. The data selected is fetched into the destination JAM variables named exactly like the database column names. Database independent or dependent aliasing can be used when the widget names and column names do not match. The format of the fetched data is determined by the properties of the destination variables.

4. JAM determines the number of rows of data that will be fetched into JAM variables by determining the number of occurrences that can be stored in each of the destination JAM variables, and using the smallest such number. Additional rows are fetched with the `DBMS CONTINUE` command.

optimizing the number of rows to fetch depends on the runtime environment

The control that the SQL executor provides over the number of rows fetched at one time is very important in two circumstances:

- Client-server applications under conditions of high network use. In this case it may be advantageous to minimize network traffic by reducing the number of rows fetched at one time.

- A database like Sybase that maintains a read lock until the `SELECT` completes. In this case, it might be critical to release the lock immediately by fetching all selected rows at one time. In practice, this means setting the number of occurrences to a large number, and then flushing the `SELECT` with the `DBMS FLUSH` command after fetching the rows. Note that after the `SELECT` is flushed, JAM is unable to fetch additional rows. This is often not a problem if the number of occurrences is large. It also simplifies application development since there is no need to program special actions to fetch additional rows with `DBMS CONTINUE`. The user can simply scroll the widgets that contain the selected rows.

JAM provides a database support routine for each supported database engine. The SQL executor uses these database routines to interact with the database engines. Some database engines support multiple connections to their databases. It is also possible to install several support routines for different database engines simultaneously, providing another approach to accessing multiple databases concurrently.

The SQL executor is discussed in Section III of this guide (page 213).

SQL Generator

The SQL generator uses the properties of screen objects to generate SQL statements, and then executes the generated statements using the SQL executor. Since the transaction manager, by default, uses generated SQL statements, there is normally no reason for you to invoke the SQL generator directly. To change the generated SQL, you can change the properties of widgets, links, and table views. To provide custom SQL, you can use the SQL executor. In fact, you cannot currently use the SQL generator except through the transaction manager. The SQL generator is discussed in detail in Chapter 18.

The SQL generator can generate `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements. It uses the database support routine to determine how to generate SQL that is correct for the target database. The transaction manager tells the SQL

generator what type of statement to generate (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) and which table view is affected. To understand how this works, you must understand how JAM uses links to represent the relationships between table views.

Table Views

A table view is a group that contains the widgets associated with a database table. Table view properties include the name of the database table, the widgets associated with the table's primary key, and whether or not the table may be updated through the table view. You can add a widget to a table view, even if the widget is not associated with a column in the database table. This facility is useful for representing data computed from data stored in the database.

Links

A link relates two table views. It is usually sufficient to say that a link relates two tables, but there may be more than one table view associated with the same table on a single screen. One table view is designated as the *parent table view*, and the other as the *child table view.*

There are two types of links, server and sequential:

Server Link

A *server link* means that the database server will be asked to join the linked tables together. In this case, the SQL generator must generate the SQL to issue the join request to the database. The table views joined together by server links are referred to collectively as a *server view.* The term server view also applies to a table view that is not connected to any other table view by a server link. When the SQL generator is asked by the transaction manager to generate SQL for the parent table view within the server view composed of server links, then the SQL generator builds a single SQL `SELECT` statement that joins all of the tables in the server view.

Sequential Link

A *sequential link* means that the two separate `SELECT` requests will be sent to the database server, and that JAM will perform the join. The transaction manager first asks the SQL generator to generate and execute the SQL statement for the parent table view. When that request is complete, the transaction manager asks the SQL generator to generate and execute the SQL statement for the child table view. The join occurs because the first request provides the information needed by the second request.

In our sample screen, the `titles` table view is joined to the `roles` table view by a sequential link. Therefore, when the transaction manager asks the SQL generator to generate a `SELECT` statement for `titles`, the SQL generator will not query the `roles` table at the same time. Conversely, the `roles` table view is joined to the `actors` table view by a server link. Therefore, when the transaction manager asks the SQL generator to generate a `SELECT` statement for `roles`, the SQL generator will query the `actors` table at the same time by creating a single `SELECT` statement that joins the two tables.

Note that the transaction manager provides the facility that enables joining of data between multiple databases. To join tables from different databases, specify a sequential link between the tables. The transaction manager will perform the join by first querying the parent table, and then the child table.

Transaction Manager

The transaction manager enables access to databases without SQL, JPL, or C programming. The behavior of the transaction manager is largely determined by the transaction models used by the application. JYACC provides one transaction model per database engine in order to enable JAM to effectively use such database engine features as connection management, cursor management, and concurrence control. The behavior of a transaction model can be changed by writing transaction manager hook functions using JPL or C, optionally along with SQL. You can even write complete transaction models. These abilities, combined with the database-specific models provided with JAM, give you the ability to use the powerful features of your database with zero or minimal programming.

Database access is achieved by creating an interactive transaction with the transaction manager. A user's notion of a transaction is that information is gathered, modified, and committed to the permanent memory of the computer. The user might abort such a transaction at any time prior to the commit. The transaction manager, via the transaction models, determines when and how database engine transaction services are used to support the interactive transaction. For example, consider the action of selecting data for possible update. Some models begin a database transaction when the data is selected. Other models begin a database transaction only after the user has requested that the changes be saved.

Do not confuse JAM's interactive transactions with database transactions. JAM does not provide database transaction management. JAM converts user requests into database transactions.

When a set of records is selected for update, the transaction manager keeps a before image of the selected data. The user may add, change, or delete data on the screen. When the user requests that changes be saved, the transaction manager determines what `INSERT`, `UPDATE`, and `DELETE` statements must be generated and executed in order to make the database reflect the changes made to the screen. The properties of the links determine, when `INSERT`, `UPDATE`, and `DELETE` statements are generated, whether the statements are applied first to the parent or child table views. This enables JAM to work with both referential integrity constraints and cascading deletes.

transaction model

In addition to describing differences between database engines, transaction models can be used to describe how the application behaves in response to various user requests. For example, invocation of stored procedures can be substituted for the transaction model default behavior of executing generated SQL. This is just one example of how user interaction can be controlled by the transaction model. Another example is that menu items and push buttons are automatically activated and deactivated, and widgets used for data entry are protected and unprotected, as a result of the state of the transaction. One of the key features of the models is that user interface code can be centralized and separated from business logic for code simplification and reduction.

To enable full customization of transaction models, all transaction models provided with JAM are provided in source form.

For details on the transaction manager, refer to Section V (page 307).

Transaction Manager Commands

To add transaction management to a screen, drag screen objects from the repository onto the screen. If these objects were imported from the database, as described later, then the transaction manager is likely to have all of the information it needs to build transactions. To enable the user to use transactions, you must attach high level transaction manager commands to push buttons, menu items, or function keys. These commands closely correspond to how users perceive transactions. The most common of these commands are summarized in Table 6 and detailed in Chapter 23.

Table 6. Common transaction manager commands.

Command	Description
`CLEAR`	Clear transaction data (i.e. data entered into widgets) from the screen. This applies only when entering data for a new record or when entering criteria for selecting records.
`CLOSE`	Abort the transaction in progress, but first prompt for and receive user confirmation.
`CONTINUE`	Fetch the next group of selected records.
`COPY`	`CLOSE` the current transaction (if any), without clearing existing data, to enable the user to create a new record from existing data.
`NEW`	`CLOSE` the current transaction (if any), and `CLEAR` existing data, to enable users to enter information to create a new record.
`SAVE`	Update the database to reflect the changes made by the user, or the new data entered by the user.
`SELECT`	Select one or more records for possible update.
`VIEW`	Select one or more records for viewing purposes only.

root table view

The transaction manager executes its commands by traversing the table views on the screen, using the specified links. It starts with the *root table view*, which is the table view that is the ancestor of all other table views on the screen. The transaction manager can affect only those table views that are descendants of the root table view. For example, when JAM imported the `roles` table earlier, JAM created a link between `roles` (the child) and `actors` (the parent). The movie screen works only if `actors` is the child and `roles` is the parent, because the root table view is `titles` and the sequential link is between `titles` (the parent) and `roles` (the child). If the `roles+actors` link were not changed, then `actors` would have no parent and would be unreachable from `titles`.

Transaction Modes

A transaction is associated with a single screen. This enables many transactions to be active at one time. For example, a user may be in the process of updating one employee's records when another employee calls to check on his insurance status. The user might interrupt work on the first employee, satisfy the second employee's request, and then return to the first employee's record. The state of a transaction is characterized by its mode. The modes are:

- initial — Transaction is inactive. The user can either issue a `NEW` command to move to new mode and begin entering data for new records, or enter select criteria and issue a `SELECT` or `VIEW` command.
- new — Data is being entered for new records to be added to the database.
- update — Existing records are being modified.
- view — Existing records are being displayed.

The modes enable the transaction manager to determine which commands are permitted at a particular instant, and to take different actions for a particular command. For example, an error is detected if `SAVE` is attempted in any mode other than new or update.

Styles and Transaction Classes

Modes also permit the transaction manager to change the appearance of objects to reflect the state of the transactions. This is managed via styles and transaction classes, both of which are created and modified with the styles editor. A style is a description of an object's appearance and behavior—for example, its color and protection. A transaction class is a pairing of styles and modes.

To minimize the work needed to build an application, JAM provides predefined styles and transaction classes, and a special class named "default". By default, JAM assigns the transaction class "default" to each object. The "default" class tells JAM to select one of the other predefined classes at runtime. Table 7 lists some of the predefined classes.

Table 7. Some predefined transaction classes and their allowable actions per mode.

Transaction Class	Default Assignment Conditions	Data Manipulation	Permitted Actions Per Mode			
			initial	new	update	view
updatable	Widget is not part of table's primary key.	Data can be changed.	focus, input	focus, input	focus	focus
primary_key	Widget is part of table's primary key.	Data cannot be changed in update mode.	focus, input	focus, input		focus
non_updatable	Widget is part of table's primary key.	Data is not updatable.	focus		focus	focus

The automatic assignment of the "default" class means that you can do significant application development without defining styles, without defining transaction classes, and without setting the transaction class property of objects.

Transaction classes also apply to push buttons and menu items. This enables JAM to automatically activate and inactivate push buttons and menu items according to the transaction mode.

Changing Model Behavior with Hook Functions

You can write transaction hook functions to customize the processing of transactions. Hook functions can be used to:

- Modify automatically generated SQL.
- Supply hand-coded SQL to replace generated SQL.
- Supply stored procedure or RPC calls to replace generated SQL.
- Change error handling.
- Change the use of cursors.
- Change the use of database transactions.

A hook function replaces part of the functionality provided by a transaction model. A model can be thought of as a collection of hook functions. JYACC supplies transaction models with JAM. These models are similar in many regards in order to promote database portability. These models are referred to as the *standard models*.

The transaction manager processes commands. It generates transaction events, in response to each command. Each event is "sent" to the table views involved in a transaction. A transaction event can be thought of as a request for a table view to do some of the processing required by a command. In fact, the events generated by

the transaction manager in response to commands are referred to as *requests* in order to differentiate them from other types of transaction events. The table view-specific processing is carried out by the transaction model associated with the table view. It is the model that knows, for example, whether or not to request generation and execution of SQL in response to an event.

The standard transaction models are complex and powerful. Most applications that use a single database can use a single transaction model throughout the entire application. When the model does not perform the needed functionality, a transaction hook function can be associated with a table view to provide that functionality. When a transaction hook function is present, the transaction manager sends requests to the hook function rather than to the model. The hook function decides whether or not the transaction manager will also send the request to the model. A transaction hook function is specified as a property of a table view. The transaction model may be specified as a property of a table view or of a screen. If no transaction model is specified, what is used is the default transaction model for the default engine at the time of the `START` command (typically when the screen opens).

Here's a sample transaction hook function. It, and other details of hook functions, is explained in greater detail in the Chapter 22. This hook function replaces automatic SQL generation with a hand-coded SQL statement.

```
proc my_better_hook( event )
if event == TM_SEL_BUILD_PERFORM
{
      dbms DECLARE :@tm_sel_cursor CURSOR FOR \
              SELECT empid, name, salary \
              FROM emp \
              WHERE name = ::name
      dbms WITH CURSOR :@tm_sel_cursor EXECUTE USING name
      return TM_CHECK
}
return TM_PROCEED
```

Completing Your Application

After your screens are designed, you will need to write control strings, JPL procedures or C functions to link your screens together into an application. In addition, the following sections list some of the areas where JPL procedures or C functions can provide specialized functionality for your application.

Displaying and Managing Screens

You display and manage screens by invoking control strings or library functions. When a screen is displayed, it is opened and then placed on JAM's *window stack*.

There is no limit (other than memory) to the number of windows that can be on the window stack. A screen is said to be displayed as a *window* if it is displayed without closing other screens on the window stack. The screen at the top of the window stack is the *active window.* A screen is said to be displayed as a *form* if all open windows are first closed. To display a screen, you can invoke any of the following control strings or library functions. Note that there are other library functions that display screens, but they are beyond the scope of this overview.

Table 8. Options for displaying a screen.

Control String	Function Called From JPL	Description
`&mywin`	`call sm_jwindow("mywin")`	Display a screen as a window. This is also referred to as displaying a *stacked window.*
`mywin`	`call sm_jform("mywin")`	Display a screen as a form.
`&&mywin`	`call sm_jwindow("&&mywin")`	Display a screen as a sibling (non-modal) window.
(none)	`call sm_setsibling()`	Open the next screen as a sibling of the current window

When a screen is displayed as a stacked window, then the user will not normally be permitted to interact with underlying windows until the stacked window is closed. To permit user interaction with underlying windows, display the screen as a sibling of the underlying window. This can be done with the `&&` notation shown in Table 8. If the screen referenced by `&&` is already a sibling of the active window, then the existing sibling is activated (i.e. moved to the top of the window stack). To open a second sibling instance of a screen already displayed as a sibling to the active window, call `sm_setsibling`, then open the screen as a sibling window.

The first screen displayed by JAM is displayed as a form. JAM maintains a form stack that contains the names of screens displayed as forms. When a form is closed, JAM removes its name from the form stack and reopens the next screen on the form stack. Before displaying a screen as a form, JAM first checks for the screen name on the form stack. If it is there, then JAM clears the form stack of all names above the name of the screen to be displayed.

You can use forms to:

- Quickly close a deeply nested set of open windows.
- Save memory by keeping track of screen names, rather than by maintaining open windows.

For details on screen management, refer to page 69.

Sharing Data Between Screens

JAM applications typically require data to be shared between screens. JAM provides several methods of sharing data, as described below.

Sending and Receiving

This technique is similar to passing parameters when calling a function. It 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. Sending and receiving are implemented by the JPL statements `send` and `receive`. There are also JAM library functions used for sending and receiving.

JAM creates a *bundle* that contains sent data. By default, the bundle is nameless and is automatically destroyed upon receipt. However, you can name bundles and preserve them after receipt so that data can be shared among several screens.

Explicit Reference

In JPL you can refer explicitly to a data item on an open window by using the notation *screen-name*`!`*widget-name*. For example, you could copy data from `empid` on the `empsrch` screen to `empid` on the active screen by writing:

```
empid = 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, JAM 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 is local to a user, but global to the application

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. In most circumstances do not use the LDB to pass data between two screens, or for other purposes where global data is inappropriate.

For details on moving data between screens, refer to page 191.

JPL and C Variables

Data can be moved between screens by saving the data in JPL or C variables, opening the destination screen, and copying the data into the destination screen. For example, in JPL you could write:

```
...
vars pass_empid
pass_empid = empid
call sm_jwindow(”empview”)
empid = pass_empid
...
```

Note that the reference to `empid` after the call to `sm_jwindow` refers to a widget on the newly opened screen, whereas the first reference refers to a widget on the original screen. The advantage of this technique over the *screen-name*!*widget-name* technique is that the source and destination screens do not need to be open at the same time.

Manipulating JAM Events

When you open and close JAM screens and move among the widgets on the screen, you move through a series of JAM events. Categories of JAM events include screen entry, screen exit, widget entry, widget exit, group entry, and group exit. You can programmatically decide what behavior occurs for each event either on a screen-by-screen basis or at the application level. To change the behavior for the entire application, you need to write and install a hook function. For more information on hook functions, refer to page 115.

Changing Object Behavior at Runtime

All JAM objects have properties that can be examined or changed at runtime. Most of these properties can be viewed in the screen editor through the Properties window; however, there are application and runtime properties that are available as well. You can access these properties through library functions or through JPL.

The following library functions allow you to examine and change screen and widget properties at runtime:

- `sm_prop_get` gets properties of the application and its components—screens, widgets, LDBs and so on.
- `sm_prop_set` sets gets properties of the application and its components.

You can also access and set these properties in JPL. For example, the following section of a JPL procedure sets the Hidden property of a widget named `title_id`:

```
title_id->hidden=yes
```

For more information about accessing JAM properties, refer to page 28 in the *Language Reference*.

Using C with Your Application

JAM provides an excellent interface to functions written in C. The interface enables C programmers to:

- Integrate third-party software with JAM applications.
- Write hook functions in C.
- Extend the functionality of JAM by writing library functions for JPL programmers.
- Write transaction models in C.
- Use JAM without JPL.

To assist C programmers, JAM's `main` function is provided in source code form in a file named `jmain.c`. Hook functions are installed in a provided source file named `funclist.c`. For details on hook functions, refer to page 115. The full set of JAM library functions is described in the *Language Reference*.

Summary

In the following chapters, the topics mentioned in this overview are explained in more detail. Additional manuals may help with certain areas like the screen editor, library functions, or configuration. If you would like to use JAM in order to apply some of the concepts presented in this overview, refer to the tutorial in the *Getting Started* manual.

If you want to look at a sample JAM application, there are two applications provided in your JAM distribution. One is `videobiz`, the sample application for a video store which is described in the next chapter. The other is JISQL, the interactive SQL editor for JDB.

Sample Application: VideoBiz

JAM is installed with a sample application—VideoBiz. It is provided to give you a look at a working JAM application. Although all JAM 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 will assist you in developing your JAM application.

VideoBiz is a database application that was built to take advantage of JAM'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 to the sample application from the 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 38 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 JAM⇒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.

For Windows

To start VideoBiz, choose the VideoBiz icon in the JYACC program group. The VideoBiz Welcome screen opens.

For Motif and Character-Mode

To start VideoBiz:

1. Know in what directory JAM is installed.
2. From your home or working directory (make sure your current directory has write-permissions) run the following script:

   ```
   . $SMBASE/samples/videobiz/vbizunix
   ```

   The required files you need locally are copied to your current directory. They include: the `videobiz` database and the styles sheet `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.

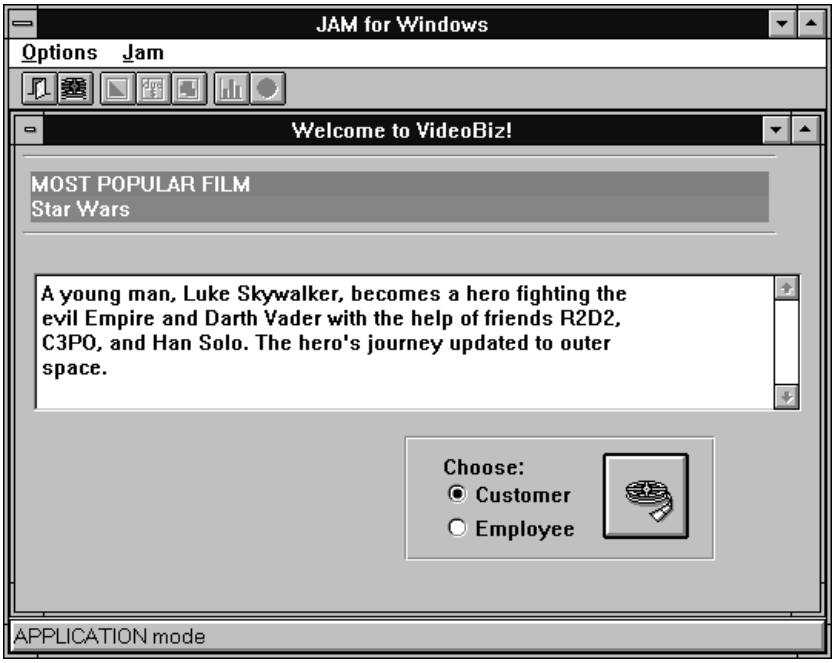


Figure 9. VideoBiz Welcome screen includes a menu bar, a graphical toolbar, two logon options, and a display of the most frequently rented video.

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 (JYACC's prototype database tool) 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. JAM'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 JAM 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 (JYACC database) 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 Jam⇒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. JAM'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 JAM menu option is also provided on the menu bar to allow you to easily access the JAM 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 JAM `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 JAM executable includes JAM/ReportWriter, edit the `videobiz.jpl` file to let JAM know this. Change the following line:

```
RW_INSTALL = 0
```

to `RW_INSTALL = 1`

The report templates are provided as JAM screens by the JAM/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?

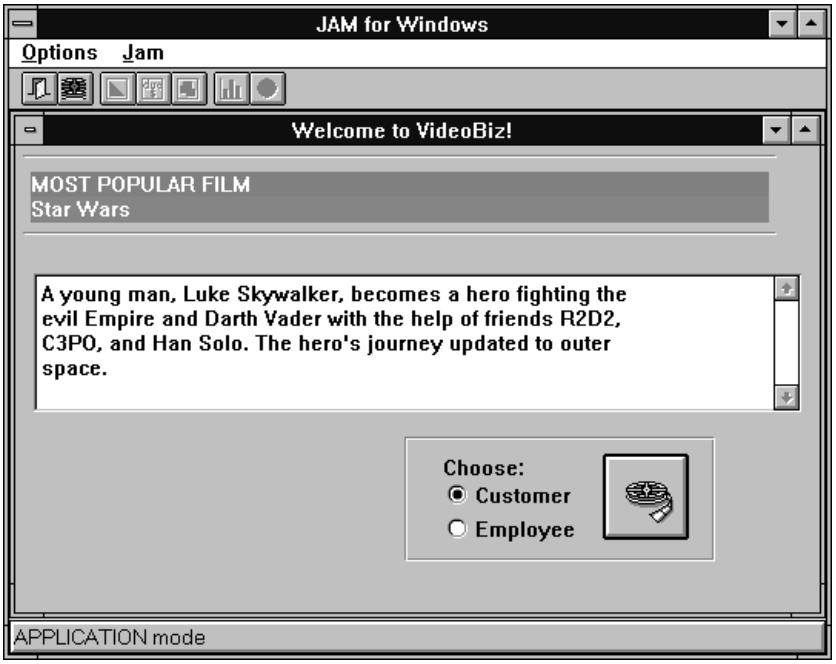
VideoBiz is a small database application that is intended for use in a video rental store. It serves three audiences and provides functions specific to those audiences:

- 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 Jam. 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 JAM menu is provided for your convenience to give you access to the JAM 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:

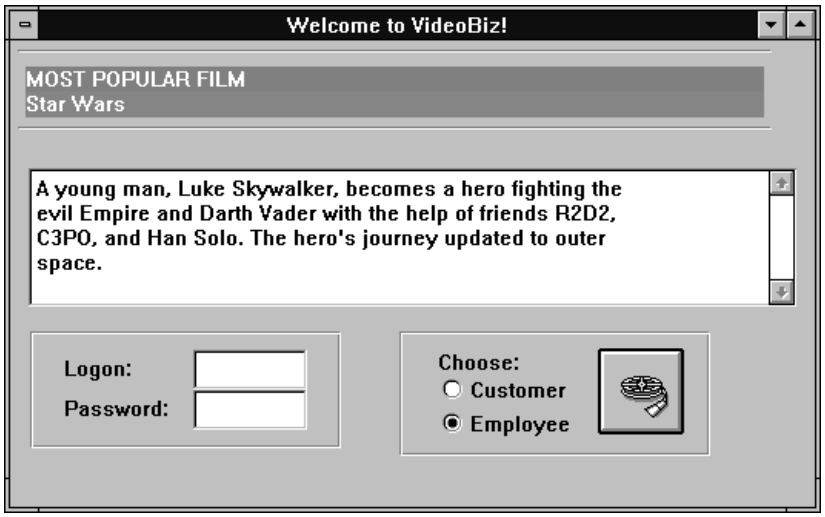
- Customer — The default radio button. When this button is selected, pressing Enter or choosing the Start button invokes the Search for a Video screen. (Under GUI platforms, the Start button displays a pixmap of a 35mm camera reel on it.)

  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.

To log into VideoBiz as an employee:

1. Choose the Employee radio button. The log on fields (Name and Password) are displayed.



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.

To exit VideoBiz

Choose Close/Quit from the application's system menu. JAM prompts you to confirm the termination of your session in VideoBiz.

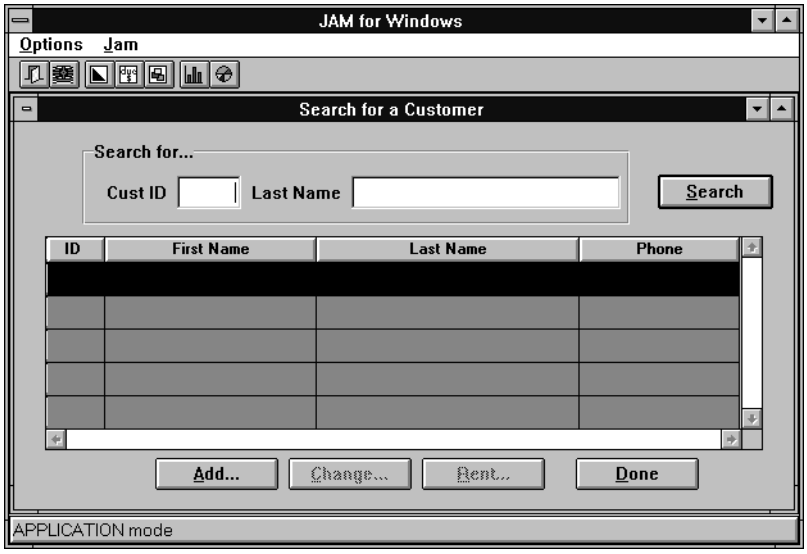
Behind the Scenes

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 JAM with the aggregate expression used in the SQL that is automatically generated on screen entry.

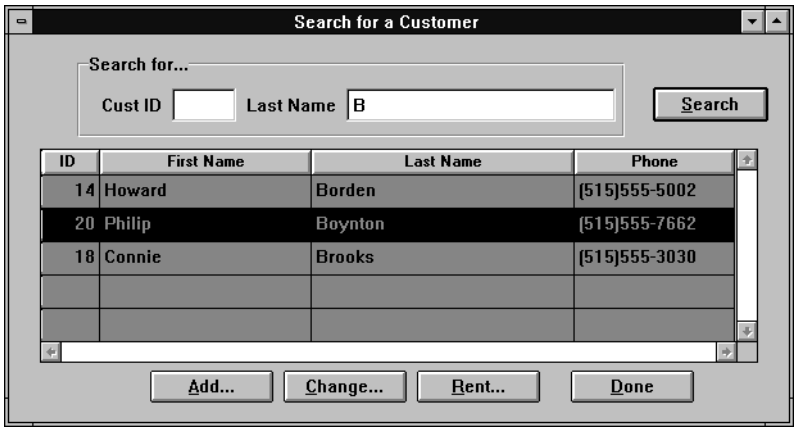
Identify the Customer

When the user successfully logs in as an employee, the Search for a Customer (`custlist.jam`) screen opens.



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 frame 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.



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 frame.

   - 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 frame. 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 frame.

   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 frame 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 frame 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 frame displays no data. The button becomes active once a customer record is selected in the grid frame. 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.

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.

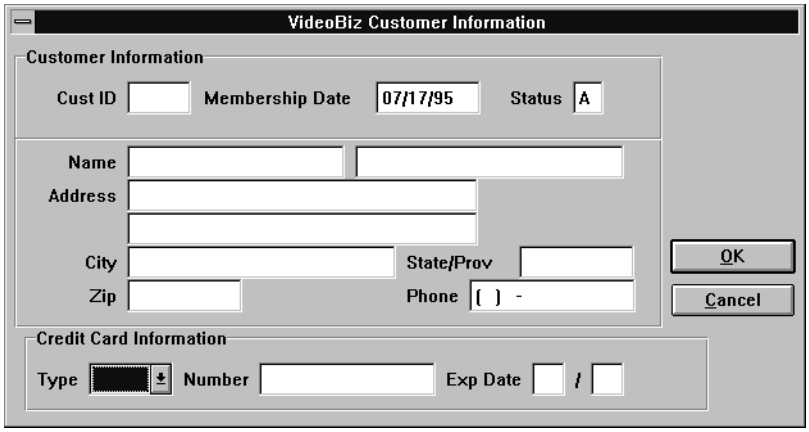


Figure 10. The Customer Information in New mode initializes the Membership Date to the system date and the Status field to A, for active. In addition, any string that the user may have entered in Last Name query field on the Search for a Customer screen is passed to this screen and is displayed in the 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.

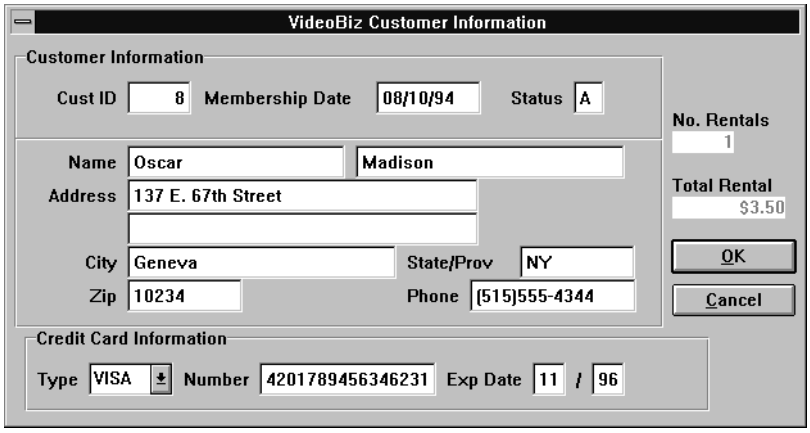
3. Choose OK to accept the information. The application assigns a customer ID number and displays a confirmation message.

   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.

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 and displays all 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.

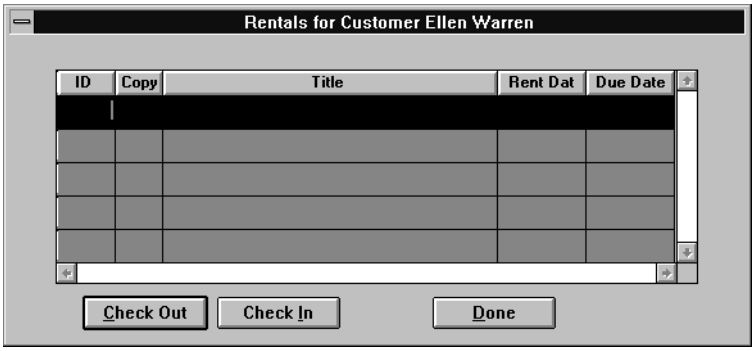
Behind the Scenes

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.



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.

To return a video:

1. Select the video from the list.

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 `cust-list.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.

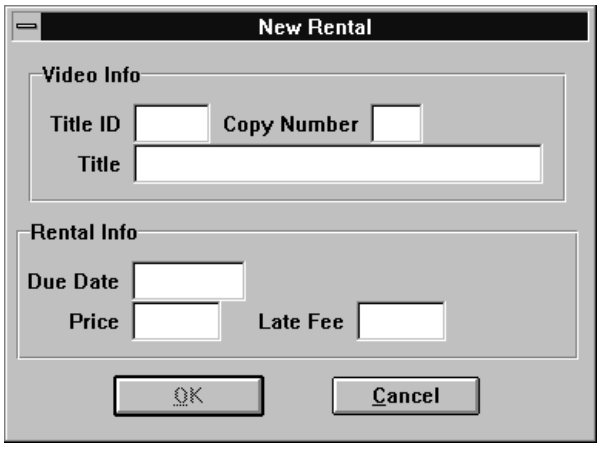
Rent Videos

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.

To rent a video:

1. From the Rentals (`rentlist`) screen, choose the Check Out button.

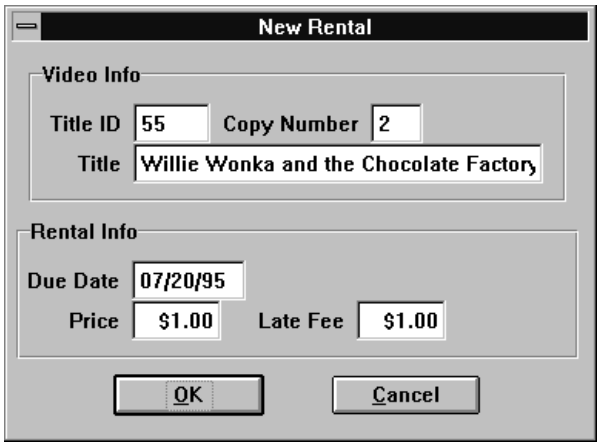
   The New Rentals (`rentvid.jam`) screen opens.



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.



   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 hook 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; most of these procedures apply the business logic required to make this screen work correctly.

Customer Profile

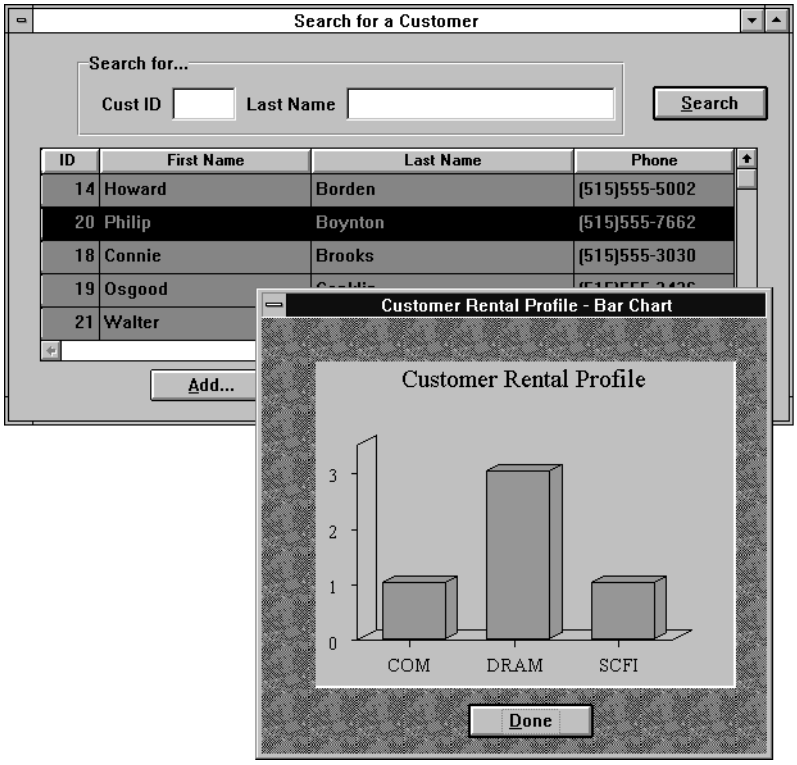
The Customer Profile options are only available to a user logged in as an Employee. A customer profile provides two different graphical representations of the types of videos a selected customer has rented.

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:

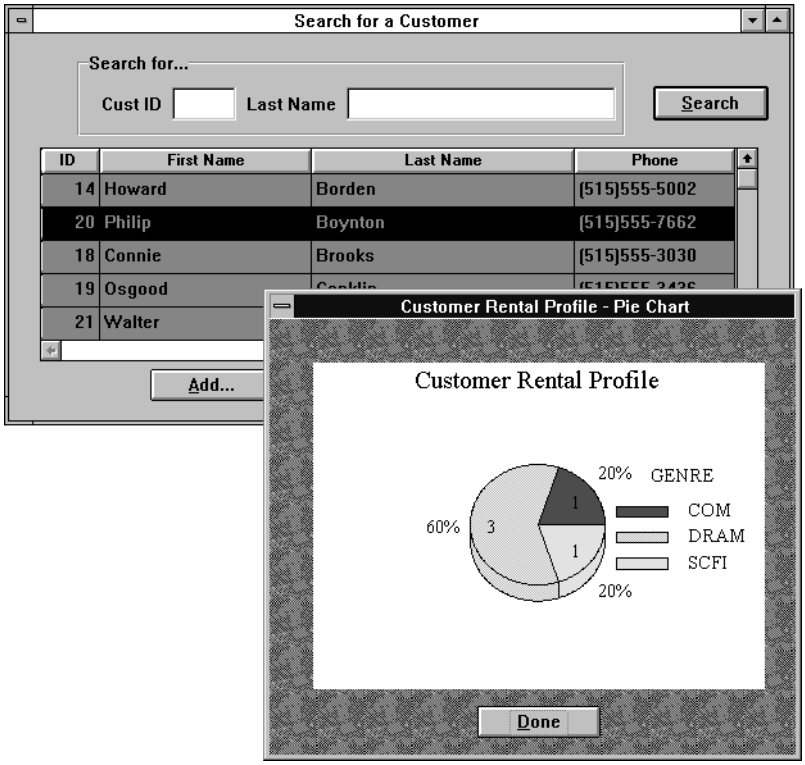
   - Choose Options⇒Profile⇒Bar Chart or 

     The selected customer's rental profile is displayed in the bar chart format:



- Choose Option⇒Profile⇒Pie Chart or 

  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. JAM automatically performs a group by category.

Video Lookup

The user can access the video lookup portion of VideoBiz by doing either of the following:

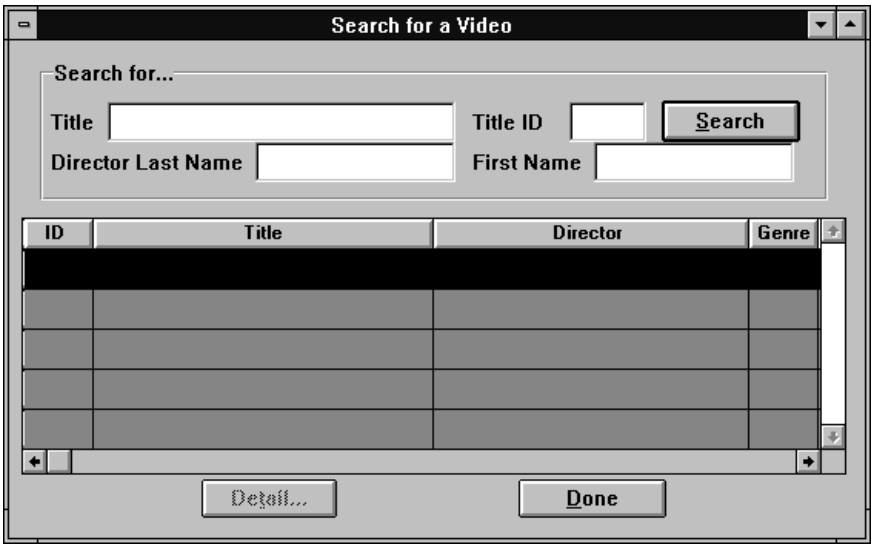
- Logging into VideoBiz as a customer (choosing the Customer radio button on the Welcome screen)
- Choosing Options⇒Video Search or 

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 frame that contains four columns for displaying query results.

The grid frame lists the video ID, movie title, director, genre, and rating. The grid frame 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.



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 frame 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 frame.

If the user chooses the Search button without specifying any search criteria, all videos in the database are displayed in alphabetical order by title.

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.

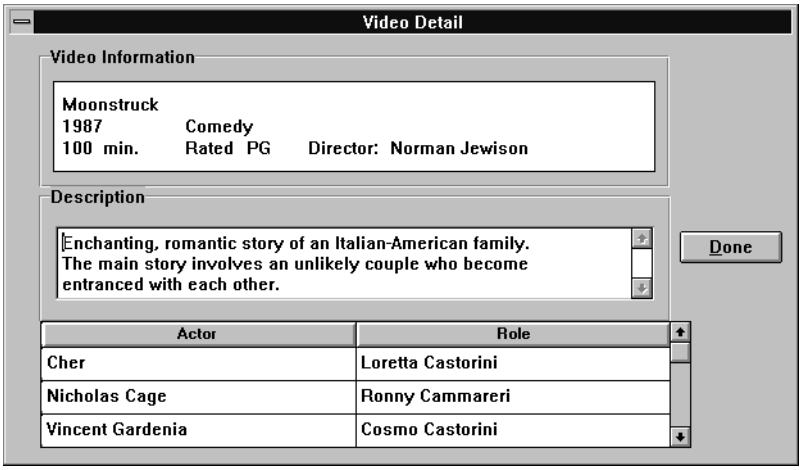
Behind the Scenes

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 frame 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 (e.g., 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 frame in the lower portion of the screen displays the actors who appear in the film and the roles they play.



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

If you have JAM/ReportWriter installed, you can access the Marketing portion of VideoBiz by choosing Options⇒Marketing from the menu bar or one of the corresponding toolbar items.

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 

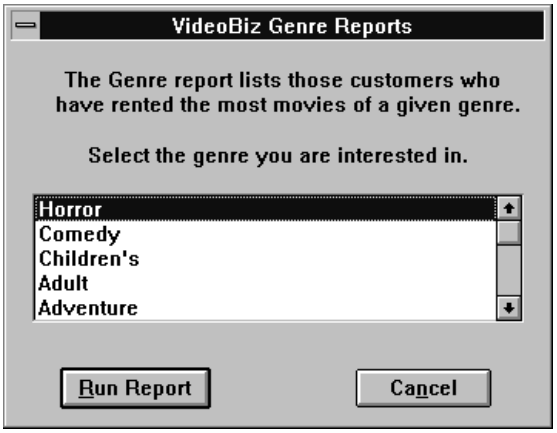
     Lists the ten most frequently rented videos.

   - 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.



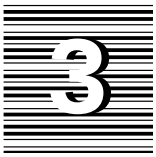
- Select a category and choose Run Report.

  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.

Data from these screens are passed to reports created with JAM/ReportWriter and output to disk files. Sample output from each report is provided and is displayed if the JAM executable doesn't have JAM/ReportWriter installed. Otherwise, the reports are actually run and displayed.

SECTION TWO

Application Building Blocks



Repository

The visual object repository is used during development to define and store the various objects needed to build your application screens. Once the repository is populated, you can easily make a new application screen 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 the objects by using inheritance. When you copy a new object from the repository, the copy of the object, 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 consists of a series of JAM screens. These repository screens can contain:

- Screen templates
- Widget templates
- Widgets imported from your database tables
- Templates for the screen wizard

Using the Repository

The repository and the ability to inherit repository properties are powerful development tools. 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

The first step is to create a repository from options found in the screen 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 page 55 in the *Editors Guide*.

Note that if the repository is named `data.dic`, JAM automatically opens it when you start `jamdev`.

Creating Repository Entries

Once the repository is created, you can populate it. A repository is like a JAM library, in that, it is a collection of screens. Each of the screens in the repository is called a repository entry.

There are several ways to create repository entries:

- Choose the New⇒Repository Entry option on the File menu.
- Save any JAM screen as a repository entry.
- Import your database tables, views or synonyms. This creates a repository entry for each database table.

Creating Repository Objects

When you import your 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 screens have complete access to all of the screen 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 screen.

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

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.

Once a screen is created, it can inherit screen properties by setting the Inherit From property to the repository entry. The dialog box asks whether you want to inherit all the screen properties. If you select Yes, all the screen properties will be taken from the template, overwriting any values you have set. If you select No, the Inherit From property will be set to the screen template but no property values will be inherited. You can then set inheritance for each property by clicking on the Inherit button while the property is selected.

Note that widgets stored on your screen template are not copied to the application screen, just the screen properties.

Note: *To define the colors for application screens, you might choose to define and edit settings in the configuration map file. Refer to the Configuration Guide for more information.*

Storing Database Information

Using the database importer in the screen editor, you can import a database table with all of its column definitions and primary and foreign key relationships into a JAM 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 entry, JAM creates a label and 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 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 screens.

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 repository entries. Repository entries created from the database import facility contain the necessary settings for SQL generation needed by the transaction manager.

Storing Widget Templates

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

You can define the widget template on a repository screen, or you can copy a widget from one of your application screens to the repository. If you copy a widget to the repository, please note that inheritance is automatically set for the widget on the application screen.

Storing Widget Definitions

In some cases, you can make repository entries based on sets of widgets used throughout an application. These sets of widgets, which might be used on several screens, have several properties that differ from the database repository entry.

First, import the database tables to the repository. Copy widgets from the database repository entry to new repository entries. 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 Screen Wizard

The first time you create a new screen with the screen wizard, JAM copies two entries into the current repository:

- `smwizard`
- `smwizis`

`smwizard` is the template for the screen itself as well as the template for several objects found on the finished screen, including the push buttons and the grid frame. Since these objects inherit from `smwizard`, you could change the properties in the repository so that every new screen made with the screen wizard would inherit the the desired settings.

`smwizis` is the template for the item selection screens. It too contains push buttons and a grid frame and could be used to modify the properties for those objects or for the screen itself.

Note that if your repository is set as read-only, the screen wizard cannot make the necessary entries in the repository. You need to speak to your system administrator about adding these entries to the repository if you want to use the screen 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, if a widget's Inherit From property contained `titles!title_id`, it indicates that the widget inherits its properties from an object whose Name property is `title_id` on the repository entry `titles`.

For the 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 child will reflect those changes when the screen is opened in the screen editor or after you update the screen using the `binherit` utility. For more information on `binherit`, refer to the next section.

If a property is inherited, it is highlighted in the Properties window of the screen editor. If you turn off inheritance for a property and later wish to reinstate it, select the property and press the Inherit button at the bottom of the Properties window.

The Inherit selection on the Options menu is available in the screen editor in order to help system performance. 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. You can activate inheritance by choosing Inherit on the Options menu, by opening the screen with the Inherit selection being active, or by saving the screen and running the `binherit` utility.

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

Updating Inheritance in Application Screens

Use the `binherit` utility to update application screens or a screen library from property values stored in the repository. `binherit` can also be used to report on the differences in the properties between the screens and the repository.

Inheritance is updated each time you open a screen in the editor and then save it. `binherit` performs this operation in batch mode, opening the specified screens and saving them.

To report or update a series of screens, use the following format:

`binherit` *[-rrepository-name] [-vlevel] [–u] filename*
[filename ...]

To report on or update members of a screen library, use the following format:

`binherit` *[-rrepository-name] [-vlevel]*
[–u] `-l`*library member [member ...]*

Square brackets indicate the optional command flags and do not need to be typed.

To obtain a brief description of available arguments and command options, type

```
binherit -h
```

`binherit` opens the specified file and looks for widgets having the Inherit From property set to a repository entry, for example, `title_id@titles`. For those widgets, it compares the inherited property values with the values in the repository. The properties that have inheritance disabled are ignored.

It also checks to see if the screen's Inherit From property is set. If it is, 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.

Arguments and Options

repository name
The name of a repository containing the entries for the specified files or library.

filename
The name of a JAM screen or library; more than one filename may be included. If the filename does not exist or is not of the correct type, it is skipped.

library
The name of a JAM screen library.

member
The member of a screen library; more than one member can be included. If the library member does not exist, it is skipped.

`-r` Specifies the name of the repository. If `–r` is not specified on the command line, `binherit` looks at the value of `SMDICNAME`. If this variable is not set, `binherit` reports an error.

`-l` Specifies the name of the screen library when updating or reporting on individual library members.

`-v` Specifies the level of reporting desired when running the utility.

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.

`-u` Update the screens as well as listing the differences.

Errors

The following list describes possible errors, their causes, and the corrective action to take:

`Not a JAM repository.`

Cause: File specified after the `-r` option was incorrect.
Action: Check the spelling and location of the specified repository.

`Unable to inherit property` *property_name* `for` *object_id*

Cause: The object listed in the Inherit From property cannot be found in the current repository.
Action: Make sure the current repository was specified.

`Unable to open JAM library.`

Cause: Unable to find the specified library.
Action: If the library is not in the current directory, include the pathname.

`Unable to open JAM repository.`

Cause: Unable to find the specified repository.
Action: If the repository is not in the current directory, include the pathname.

`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.

Screen Management

A JAM application is largely composed of screens—screens that you open as forms or windows, save as repository entries, and use to merge data into other screens or widgets. The *Editors Guide* shows how to build screens and tie them together. This chapter discusses the underlying architecture of screens and how JAM manipulates them. It also discusses runtime options that are available to change screen properties.

Forms and Windows

User interaction with a JAM 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. JAM 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. JAM 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. JAM 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.

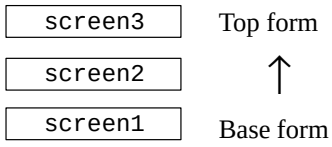
Forms and the Form Stack

JAM 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. JAM 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 JAM's send and receive commands and library functions. For more information on LDB processing and send/receive facility, refer to page 191.

JAM 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, JAM 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, JAM 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 `OK_CLOSELAST`, the application terminates; otherwise, the application continues to run without an open screen.

Windows and the Window Stack

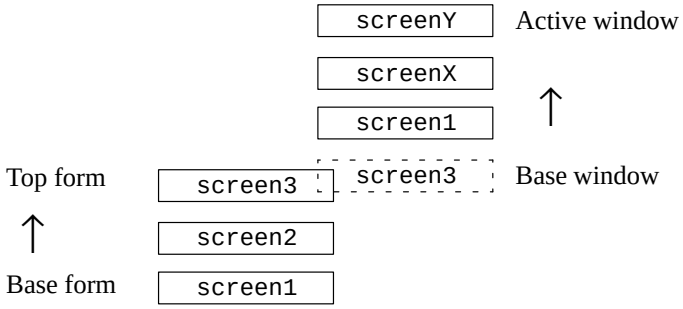
The top form always has its own window stack, in which it serves as the base window. Only the top form maintains a window stack. The window stack remains

in memory until JAM 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

JAM stacks windows in last-in/first-out 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—JAM 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.

JAM 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, JAM 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.

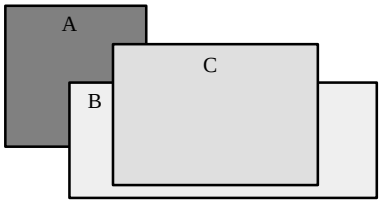
Note: *A window cannot directly open any screen as a sibling that is already a sibling. If you want to open multiple instances of the same screen as sibling windows, call* `sm_setsibling` *to force sibling status onto the next screen opened as a window. You can also reset an existing window's* `sibling` *property.*

Window Stack Manipulation

JAM 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. JAM 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`.
- `sm_sibling` assigns sibling status between the current window and the window immediately below it in the window stack. Use this function to group as siblings any number of windows that are contiguous in the window stack. You can also use this function to remove sibling status from windows.

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);
```

Opening Screens

Applications typically let users open screens by pressing a key or choosing a menu item or a selection-type widget such as a 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

You can also use JAM 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.

Note: *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.*

Screen Display Defaults

Unless otherwise specified, JAM tries to display the entire screen. If a screen is opened as a form, JAM 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, JAM tries to leave the calling screen's last cursor position visible. Note that in GUI environments, the displayed form always leaves the menu bar visible.

If the screen size exceeds the physical display area, JAM creates a viewport—that is, a window with scroll bars that displays portions of the screen. 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. Note that JAM 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.

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*, JAM 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 detailed information about control strings, refer to page 109.

usage in runtime functions

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 );
```

Screens and Viewports

JAM 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, JAM displays the screen in a *viewport* with vertical and horizontal scroll bars, so users can scroll out-of-view data into view. JAM also scrolls screen data when the user tabs into an out-of-view field to make that field visible.

viewport size and display modes

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.

Closing Screens

By default, EXIT 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 attaching it to a physical key, menu item, or push button.

You can also 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.

Screen Properties

When you create a screen, JAM 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, JAM 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 removing its Inherit From property. If inheritance remains on, the repository can be used to control screen properties at runtime.

You can get and set all screen properties at runtime through JPL. For example, this statement gets the `title` property for screen `vidlist.jam`:

```
cur_title = @screen(”vidlist.jam”)–>title
```

For more information about accessing JAM properties, refer to page 28 in the *Language Reference*.

Field Management

JAM 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 check list, and determine whether data has changed or passed validation. JAM identifies widgets that can be thus accessed and manipulated as fields, in contrast with other widgets that are static in nature, like lines, boxes, and static labels.

This chapter shows how to:

- Identify and access widgets programmatically.
- Get information about widgets and their data.
- Manipulate widget properties and data.

All functions described in this chapter are documented in the *Language Reference*; refer to that manual for the syntax and specific behavior of each function.

Field Names and Numbering

Most runtime functions let you identify fields by name or by number. In JPL, you can always refer to fields by their name or number. You can also refer to named fields on other screens. For more information on referencing fields in JPL, refer to page 24 in the *Language Reference*.

Field numbers are assigned automatically when you add a field to a screen, and are reassigned whenever the field position changes. JAM numbers fields as follows:

- Fields are numbered sequentially from left to right, and top to bottom—as `Field #1`, `Field #2`, and so on.
- All onscreen occurrences of an array, called *elements*, are numbered. The number of the first element in an array, or the array's *base field*, is the number by which the field as a whole is identified. Field numbers of subsequent elements can be viewed only in the Widget list.

Note: *Elements in an array might not be numbered contiguously, depending on whether other fields are positioned on the array's right side.*

Because a field's position determines its number, changing design considerations and runtime repositioning can make referencing fields by number problematic. Names provide maximum control of runtime field behavior; they are also easier to identify in your code. Additionally, JAM requires named fields for these reasons:

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

Naming Fields

A widget name can be up to 31 characters long and can start with an alphabetic character, an underscore, a dollar sign, or a period. Names are case-sensitive; thus, JAM identifies `city` and `City` as two unique 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.

Widgets that are created as the result of importing database tables are automatically named.

Identifying Fields

JAM provides several properties, accessible through JPL or library functions `sm_prop_get` and `sm_prop_set`, that contain a field's number or name. In JPL, these properties are:

- `fldnum` — A property of the current screen and of widgets. As a screen property, it is set to the number of the field where the cursor is currently positioned. As a widget property, it is set to the field number of a field specified by name, or the base field number of an array specified by name.
- `name` — As widget property, set to the name of the number-specified field.

Arrays

JAM identifies an array as any field that can contain one or more occurrences of data. In this sense, most JAM widget types can be regarded as arrays. Typically, however, the term array refers to three widget types: multiline and single-line text, and list boxes. 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.

JAM allocates memory only for occurrences that have data; trailing occurrences that are empty are discarded. JAM maintains information about the number of occurrences allocated for an array and the offset of occurrences within an array's elements.

Groups

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

Two 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 JAM library functions to

manipulate group fields by converting group references into field references. For example, to access text from a specific field 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 field and the field's offset within the group.

Getting Data and Properties

JAM library functions let you obtain the data in a field or its occurrences; they also let you ascertain the field's current property settings.

Getting Field and Array Data

The following functions copy data from fields and arrays:

- `sm_getfield` copies data from the specified field or occurrence to the supplied parameter. JAM strips leading or trailing blanks.
- `sm_fptr` returns the contents of the specified field. JAM strips leading or trailing blanks.
- `sm_ww_read` copies word-wrapped text from a multiline text widget into a string buffer.
- `sm_ww_write` puts text into a wordwrap field.
- `sm_ww_length` gets the number of characters in a word wrap field.
- `sm_dblval` returns the contents of the specified field as a real number.
- `sm_intval` returns the integer value of the data contained in the specified field, including its sign. All other punctuation characters are ignored.
- `sm_lngval` returns the contents of the specified field as a long integer. It recognizes only digit characters and a leading plus or minus sign.

You can also get information about the data in a field with these functions:

- `sm_dlength` returns the length of the data in the specified field or occurrence of a field. The length includes any data that is shifted offscreen and therefore out of view. The length excludes leading blanks in right-justified fields, and trailing blanks in left-justified fields.

- `sm_is_no` and `sm_is_yes` compare the first character of the data in the specified field 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 field's value is null or not. This function checks whether a field's Null Field property is set to Yes; if it is, `sm_null` gets the field's null indicator and compares it to the field's value.

Getting Properties

You can access all widget properties at runtime through JPL. For example, this `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, refer to page 28 in the *Language Reference*.

Checking Validation

JAM maintains a bit for each field and group called the validation bit that indicates whether or not the field or group has passed its edits. The validation bit is initially cleared when a screen is displayed. It is cleared again each time the content of the field is changed. It is set each time the field passes its validation tests; for example when the user tabs out of the field.

JAM also maintains a modified data tag bit (MDT) for each field and group. The MDT bit is cleared when the screen is displayed—after the screen's entry function is called—and is set when the content of the field or group is changed. JAM never clears a field's MDT after the screen is displayed, but it can be cleared by resetting its `mdt` property to `PV_NO`. You can also check and modify a widget's `valided` property.

JAM performs validation processing of a field or group without regard to the setting of the field's or group's validation bit. If validation requires significant processing such as a database lookup, then the validation bit should be tested by the validation function and unnecessary validation processing should be avoided. In addition the MDT can be used in conjunction with the validation bit to prevent unnecessary validation of fields and groups that were populated with valid data from the LDB by a screen entry function or via initial data and have not changed since the screen was displayed.

You can access a widget's MDT settings through its `mdt` and `valided` properties. You can also use the following functions to check and reset MDT and validation bit settings for all fields:

- `sm_cl_all_mdts` clears the MDT bits of all fields and occurrences .
- `sm_tst_all_mdts` tests the MDT bits of all on- and offscreen occurrences of all fields on the current screen. If it finds an occurrence with its MDT bit set, 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 `sm_bitop`.

Getting and Setting Selection Group Data

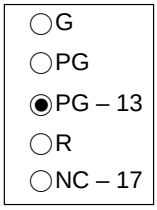
JAM has a set of functions that let you check the current selection or selections within a selection group, and change the selections.

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.

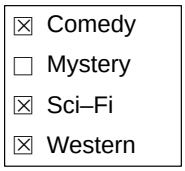
If you call `sm_n_getfield` on a radio button group that allows one selection, the buffer that you pass into this function gets the group occurrence number of the selected item. For example, the radio button group `rating` has the third occurrence, `PG–13`, selected:



Given this selection, the following call to `sm_n_getfield` puts the string `"3"` into the string buffer pointed to by `buffer`:

```
retvar = 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—JAM 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`.

JAM sees a group's value as an array whose elements contain the offsets of the selected items. Thus, JAM 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, JAM first deselects the current selection before it selects the specified group occurrence. For more information about selection widgets, refer to page 199 in the *Editors Guide*.

To deselect an occurrence,call `sm_deselect`.

Changing Widget Data and Behavior

This section outlines the various runtime options provided with JAM library functions to change widget data and behavior.

Writing Data to Fields

The following functions let you move data directly into fields:

- `sm_putfield` moves the supplied string into the specified field. If the string is too long, JAM truncates it without warning. If the string is shorter than the destination field, JAM blank fills it according to the field's justification. If the data is a null string, JAM 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 field, `sm_ww_write` breaks the word one character before the end of the field, appends a space, and wraps the rest of the word on the next line.

- `sm_dtofield` converts a real number `value` to user-readable format as specified by `format`. It then moves this value into the specified field with a call to `sm_amt_format`. If the `format` string is empty, JAM determines the number of decimal places from a data type edit, or from a currency edit. If neither exists, it uses two decimal places.

- `sm_itofield` converts the supplied value to a string and places it in the specified field.

- `sm_ltofield` converts a long integer passed to user-readable form and places it in the specified field.

- `sm_amt_format` writes data to a field, first checking whether the field has a currency edit. It 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 Field Data

Use the following functions to clear data from fields and arrays:

- `sm_cl_unprot` erases onscreen and offscreen data from all fields that are unprotected from clearing (`CPROTECT`). Date and time fields that take system values are reinitialized. Fields with the null edit are reset to their null indicator values.

- `sm_clear_array` clears all data from the array that contains the specified field. 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.

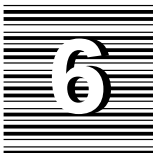
Inserting and Deleting Occurrences

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, JAM 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, JAM 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.



Menus and Toolbars

Menu bars, pop-up 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, pop-up 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, pop-up menus, and toolbars, unless otherwise noted.

JAM menu definitions are saved in menu scripts. When you save a menu through the menu bar editor, JAM saves the menu and its submenus to a binary script. At runtime, JAM 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 pop-up 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`, JAM 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 JAM 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. JAM 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, JAM writes those changes to the same memory and immediately propagates them 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. For more information on installing menus with shared and unique content, see pages 90 and 91.

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, JAM finds it in the specified script and reads its definition. If the menu contains external references—the menu is defined in another script—JAM resolves these; it then makes the menu available for display.

Except for Motif versions, JAM 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, JAM displays only the menu bar and toolbar of the active screen. 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 three 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 invoked 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.

- *Field scope*. A menu that is installed with a widget displays as a pop-up 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 Pop-up Menu property. You can also enter a menu name in the screen's Pop-up Menu property.
- Call the library function `sm_menu_install`. You must use this function to install menus at application scope.

When a screen opens, JAM 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, JAM creates a toolbar from the images associated with those items and displays it below the menu bar. JAM also uses the Menu Name property for the screen's pop-up menu.

At screen open, JAM also checks the Menu Pop-up property of each widget; JAM installs each menu specified by a widget at field scope and displays it as a pop-up 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, JAM removes it. If the same menu is already installed from the same memory location, JAM 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. JAM 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 pop-up 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 call `sm_menu_install` at screen entry with arguments of `MNS_SCREEN` and `MNL_SCREEN`.

Similarly, you can make sure that widgets have unique copies of the same pop-up 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, JAM 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, JAM 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, JAM continues the search among the scripts loaded into application memory. If no menu is found in either memory location, JAM 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 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.

pixmap properties

If a menu item is set to display on a toolbar, you should set its pixmap properties—`MNI_ACT_PIXMAP`, `MNI_ARM_PIXMAP`, and `MNI_INACT_PIXMAP`—to determine the item's display in its active, armed, and inactive states. JAM for Windows uses MFC to control toolbar display, which sets the item's inactive and armed display from the `MNI_ACT_PIXMAP` (active) property; consequently, Windows applications ignore the other two pixmap properties.

For more information about setting pixmap properties, refer to page 224 in the *Editors Guide*.

tooltip display

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 setting it to `TOOLTIP_ON` (the default) and `TOOLTIP_OFF`, respectively.

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

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

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

Changing Menus at Runtime

JAM 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 JAM library functions.

get properties

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.

set properties

`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 in `funclist.c`, which you can use to modify menu items in JPL modules. Refer to page 385 in the *Language Reference* for a list of these functions.

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, JAM 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. JAM updates the menu display at the next delayed write.

Deinstalling and Unloading Menus

Menus and their scripts remain in memory until JAM frees their memory location—for example, when a screen with its own menu is removed from the form or window stack. JAM 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. JAM 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, JAM removes them immediately. If a menu is referenced as an external menu, JAM displays an empty menu in its place.

Invoking Pop-up Menus

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

- If a field has focus, `sm_popup_at_cur` displays the first menu that it finds from the following:
  1. The pop-up menu installed for the field.
  2. The menu installed for the screen.
  3. The application-level menu.
- If the screen has focus, `sm_popup_at_cur` displays the first menu that it finds from the following:
  1. The menu installed for the screen.
  2. The application-level menu.

invoking pop-up menus from the keyboard

You can let users invoke pop-up menus from the keyboard with `sm_popup_at_cur`. For example, the following control string assignment lets the user invoke a pop-up 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. 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. Refer to page 385 in the *Language Reference* for a list of these functions.

Using the m2asc Utility

JAM's `m2asc` utility lets you convert binary menu files to ASCII and vice versa. `m2asc` has this syntax:

`m2asc –a` *[* `–fv` *] [* `–i` *include-file] ascii–file mn–file [mn–file ...]*

`m2asc –b` *[* `–fv` *] ascii–file [ascii–file ...]*

`m2asc –c` *[* `–fv` *] v5-mn-file [v5-mn-file ...]*

–a
Convert binary files to ASCII.

–b
Convert ASCII files to binary.

–c
Convert JAM 5 binary files to JAM 6 binary.

–f
Allow the output file to overwrite an existing file.

–i*include-file*
Include the specified file at the beginning of ASCII output.

–v
Generate a list of files as they are processed.

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.

Keywords

Each component of a menu definition is identified by a keyword in Table 9 and, optionally, a unique name. In some cases, JAM uses these names to resolve references—for example, given a submenu item that sets its `SUBMENU` property to `myEditSub`, at runtime, JAM 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.

Table 9. ASCII menu keywords

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`	The 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 `menu2asc` converts the ASCII file to binary format, it saves each script to its own file.

Keyword	Description
MENU	Starts a menu or submenu definition. All keywords that follow MENU identify the menu's items.
SEPARATOR	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 platform—for example, under Windows, the Windows menu with Arrange Icons, Tile, and Cascade. Applications running on character-mode, Macintosh, and Presentation Manager platforms ignore this item.

Menu Properties

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
```

Table 10 shows all ASCII menu property mnemonics and their valid values. For more information about these properties, refer to the *Editors Guide*.

Table 10. Menu properties and valid assignments

Property	Values
ACCEL	An accelerator string that specifies the keyboard equivalent for selecting this menu item.
ACCEL–ACTIVE	A value of PROP_ON or PROP_OFF specifies whether the menu item accelerator is active.
ACTIVE	A value of YES or NO allows or disallows user access to this menu item. If ACTIVE=NO, the menu item is greyed out.
ACTIVE_PIXMAP*	The name of an image file whose contents are shown for an active toolbar item—that is, accessible but not pressed. Refer to page 225 in the *Editors Guide* for valid file types, and for information about path and extension options.
ARM–PIXMAP*	The name of an image file whose contents are shown for an armed toolbar item—that is, in its pressed state. If this property is blank, Motif uses the MNI_ACT_PIXMAP property for the item's armed state. Windows uses a modified version of the Active Pixmap property to display a toolbar item's armed state and ignores this property.
CONTROL	A 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	A value of YES or NO specifies whether to find this menu's definition in another menu script. External references are resolved at runtime only.
EXT–HELP–TAG	A string expression that specifies the help text to invoke for this item.

* Ignored in character-mode.

Property	Values
INACTIVE–PIXMAP*	The name of an image file whose contents are shown for an inactive or unavailable (grayed) item. If this property is blank, Motif displays an empty toolbar item. Windows uses a grayed version of the Active Pixmap property to display a toolbar item's inactive state and ignores this property.
INDICATOR	A value of YES or NO specifies whether to show the toggle indicator.
IS–HELP	A value of YES or NO specifies whether to display this item as the rightmost item on the menu bar.
LABEL	A string expression to display as the menu item's label. To specify a keyboard mnemonic for a menu item, place an ampersand (&) in front of the desired character.
MEMO	A string expression for the Memo Text property.
MNI_ORDER*	The order in which this item appears on the toolbar. The default value is 100. You can enter any value between 0 and 200, inclusive. If all toolbar items are set to the same value, they appear in the same order as they do in the menu.
SEP–STYLE	The style used by item separators, specified by one of these values: SINGLE DOUBLE DOUBLE–DASHED SINGLE–DASHED ETCHED–IN ETCHED–OUT ETCHED–IN–DASHED ETCHED–OUT–DASHED NOLINE MENUBREAK
SHOW–ACCEL	A value of YES or NO specifies whether the menu item displays the accelerator key next to the label.

* Ignored in character-mode.

Property	Values
STAT–TEXT	A string expression to display on the screen's status line for this item.
SUBMENU	Name of submenu to invoke when this item is selected.
TEAR	A value of YES or NO enables or disables this submenu as a tear-off menu.
TITLE	A title to display with tear-off submenus.
TM–CLASS	Transaction manager property. Refer to page 285 in the *Editors Guide* for more information and valid arguments.
TOOL–TIP*	The 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. Table 11 shows which properties are valid for each component:

Table 11. Valid properties for menu definition components

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

Sample Output

The following menu script is the ASCII output of a truncated version of the menu bar used by JAM's screen editor. The main menu contains three items: File, Edit, and Help.

```
FILE:semain

MENU:sm_se_main_menu
  TEAR=NO
  EXTERNAL=NO
  ACTIVE=YES
  INDICATOR=NO
  SHOW-ACCEL=YES
  SEP-STYLE=SINGLE

SUBMENU:sm_se_file
  LABEL=&File
  SUBMENU=sm_se_file_menu
  IS-HELP=NO
  EXT-HELP-TAG=basicFilemenu
  STAT-TEXT=File Operations

SUBMENU:sm_se_edit
  LABEL=&Edit
  SUBMENU=sm_se_edit_menu
  IS-HELP=NO
  EXT-HELP-TAG=basicEditmenu
  STAT-TEXT=Editing 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

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_screen
  LABEL=&Screen...
  CONTROL=^filemenu open screen
  IS-HELP=NO
  EXT-HELP-TAG=FileOpen
  SHOW-ACCEL=YES
  DISPLAY-ON=BOTH
  STAT-TEXT=Opens new screen from a file
  ORDER=12
  ACTIVE-PIXMAP=open-act
  INACTIVE-PIXMAP=open-dis
  TOOL-TIP=Open

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_edit_menu
  TEAR=NO
  EXTERNAL=NO
  ACTIVE=YES
  INDICATOR=NO
  SHOW–ACCEL=YES
  SEP–STYLE=SINGLE

SEP:sm_se_edit_sep0
  SEP–STYLE=SINGLE
  DISPLAY–ON=TOOLBAR
  ORDER=30

ACTION:sm_se_edit_cut
  LABEL=Cu&t
  CONTROL=^(^cut)save_state 2
  ACTIVE=NO
  IS–HELP=NO
  EXT–HELP–TAG=basicEditmenu
  ACCEL=Ctrl–X
  SHOW–ACCEL=YES
  DISPLAY–ON=BOTH
  STAT–TEXT=Cuts current selection to clipboard
  ORDER=21
  ACTIVE–PIXMAP=cut–act
  INACTIVE–PIXMAP=cut–dis
  TOOL–TIP=Cut

ACTION:sm_se_edit_copy
  LABEL=C&opy
  CONTROL=^copy 1
  ACTIVE=NO
  IS–HELP=NO
  EXT–HELP–TAG=basicEditmenu
  ACCEL=Ctrl–C
  SHOW–ACCEL=YES
  DISPLAY–ON=BOTH
```

```
  STAT–TEXT=Copies current selection to clipboard
  ORDER=22
  ACTIVE–PIXMAP=copy–act
  INACTIVE–PIXMAP=copy–dis
  TOOL–TIP=Copy

ACTION:sm_se_edit_paste
  LABEL=&Paste
  CONTROL=^(^paste)save_state 5
  ACTIVE=NO
  IS–HELP=NO
  EXT–HELP–TAG=basicEditmenu
  ACCEL=Ctrl–V
  SHOW–ACCEL=YES
  DISPLAY–ON=BOTH
  STAT–TEXT=Copies contents of clipboard to current screen
  ORDER=23
  ACTIVE–PIXMAP=past–act
  INACTIVE–PIXMAP=past–dis
  TOOL–TIP=Paste

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

SEP:sm_se_hl_sep1
  SEP–STYLE=SINGLE
  ORDER=190
  DISPLAY–ON=BOTH

ACTION:sm_se_hl_about
  LABEL=&About JAMDEV ...
  CONTROL=^sm_message_box( \
    ”JAM Version 7.0%NCopyright 1994–1995%NJYACC Inc.”, \
    ”About JAMDEV”,0,””)
  STAT–TEXT=Tells about this version of JAM
```

Control Strings

Control strings specify actions that you associate with function keys, push buttons, list boxes, and menu items. Control strings can perform these tasks:

- Open a screen as a form or window.
- Execute a function.
- Invoke an operating system command.

Associating Control Strings with the Application

A JAM 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. For example, a push button widget can exit the current screen if its Control String property specifies 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 lists JAM logical keys and their control strings. For example, the following control string list lets users open two screens as windows through the logical keys F1 and F2, and leave the current screen through EXIT:

```
F1     = &custInfo
F2     = &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

JAM 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 12 summarizes these leading characters and actions.

Table 12. Control string types and leading characters

Character	Action	Example
None	Open screen as a form.	`mainmenmu`
`&`	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 program	`!ls "*.jam"`

The following sections explain each type in detail.

Opening Screens

A control string can open a screen as a form, as a stacked window, or as a sibling window. Control strings that open screens have the following syntax:

[lead-char] [`(` *viewport-args* `)` *]* `screen-name`

If you omit *lead–char*, JAM 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 page 69 for more information about how JAM manages screens as forms and windows.

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, viewport's dimensions, and 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.jam` as a sibling window whose viewport is two rows higher and two rows left of the current screen's viewport:

```
&&(-2,-2)newWindow.jam
```

If the window does not fit on the display at the specified location, JAM adjusts it as needed. JAM 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 13 contains several examples of control strings that open screens.

Table 13. Control strings that open screens

Control string	Action
`mainmenu`	Open `mainmenu` as a form at the physical display's top left corner.
`&(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:

^[(target-string [; target-string])] func-name [(arglist)]

func-name
The name of an installed or built-in function or a JPL procedure or module. An installed function can be one of JAM's library functions or your own. For information about function installation, refer to page 119. Built-in functions are preinstalled in JAM and begin with the prefix `jm_`. Built-in functions are described in Chapter 4 of the *Language Reference*.

JAM looks first among the installed functions for *func-name*, then among the JPL procedures modules. For detailed information about this search algorithm, refer to page 17 of the *Language Reference*.

arglist
One or more arguments to pass to parameters in *func-name*. Arguments for installed functions—JAM 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.

target-string
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

JAM compares *func-name*'s return value to each *test-value*, reading from left to right. If it finds a match, it processes the specified control string. If you omit a test value, JAM 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
```

JAM 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, JAM executes `cleanup`. When `cleanup` returns, JAM calls the built-in function `jm_exit`.
   - If `process` returns 1, JAM opens the `welcome_scr` screen.

Table 14 shows several control strings that call functions:

Table 14. 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_n_ascroll("IDs",1)`	Execute the library function `sm_n_ascroll`.
`^jm_exit`	Execute the built-in function `jm_exit`.

Invoking Operating System Commands

A control string that starts with an exclamation point (`!`) temporarily passes control to the operating system. At runtime, JAM 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, JAM 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(`:`). JAM's colon preprocessor expands colon-prefixed variables to their literal values before passing the string to the operating system.

Table 15 shows several operating system control strings:

Table 15. 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`.

Hook Functions

Hook functions are called at specific events during program execution. JAM identifies each stage of program execution as one type of event or another. For example, JAM identifies all screen entries as one event type, and all field exits as another. You can write hook functions for each of these events in any third-generation language that JAM supports. This chapter shows how to write hook functions in C and install them in your application.

JAM recognizes many different stages of program execution as distinct events that can invoke hook functions. Events for which hook functions are commonly written include:

- Screen entry and exit.
- Field entry, exit, and validation.
- Menu item, push button, or function key selection.

Through JAM's screen editor, you can specify many of the hook functions used by your application. For example, each screen can have its own entry function, which is specified on the screen's properties window, each field its own exit function, which is specified on the field's properties window, and so on. If the function is

installed, JAM finds the function at the appropriate stage of program execution and executes it. For example, if you specify a screen entry function for a screen, JAM invokes that function when the screen opens.

Installed hook functions can also be called from JPL procedures.

You can also install functions that JAM always executes whenever events of a certain type occur. Each function is identified with a single event type. For example, you can install a screen function that JAM executes on entering or exiting all screens.

All hook functions must be installed in the application so that JAM can find and execute them at the proper time. Many commonly used JAM library functions are already installed for you for immediate access during the design process. Installation procedures and options are discussed later in this chapter.

Hook Function Types

Hook 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* hook functions are functions explicitly called from a JAM component, such as a widget or screen, or a JPL module.
- *Automatic* hook functions are functions that execute on all occurrences of an event type. These functions are never explicitly called in the application code or screens; instead, JAM calls them automatically at the appropriate stage of program execution.

Demand Hook Functions

A demand hook function can be called by name from any component of a JAM application: groups, screens, menu items, logical keys, and JPL modules. Except for external JPL modules, hook function calls are stored with the application's screens and can be edited only through the screen editor. For example, each field's properties window lets you specify entry, exit, and validation functions.

Demand hook functions usually perform tasks that are specific to their callers. For example, you might create an exit function for a field whose data requires special processing. You can then specify this function through the field's Exit Function property. At runtime, JAM invokes the function when the cursor exits the field.

You can also write demand hook 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; JAM executes the JPL code if no C function of the same name is installed. For more information about JPL, refer to page 3 in the *Language Reference*.

Automatic Hook Functions

Automatic hook functions are functions that 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 field, screen, or other application component. Automatic hook functions cannot be written in JPL. However, you can call a JPL procedure from an automatic hook function through `sm_jplcall`.

In general, an application can have only one automatic hook 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 functions are the exception among automatic functions: you can install multiple timeout functions, where each one is called when its own timeout occurs.

Note that JAM can call automatic and demand hook functions for the same object. For example, on screen entry, JAM always calls the automatic screen function if one is installed. If a screen also specifies an entry hook function, JAM then calls and executes this function, too.

Standard versus Non-standard Arguments

JAM automatically supplies a fixed number of arguments for all hook function types except those that are installed as type `PROTO_FUNC`, or *prototyped* functions. Arguments that are automatically supplied by JAM 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 JAM.

You can also write functions whose arguments are explicitly supplied by the application. These functions must be installed as prototyped functions. JAM 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 hook 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.

Note: *For greater efficiency, prototyped function declarations should be #included in* `funclist.c`.

Preparing Hook 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-JAM 6 applications.

Function Name

The first value of a `fnc_data` structure 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 hook function type allows installation of only one function, supply 0.

Function Address

The second value of a `fnc_data` structure is the address of the function—that is, its C identifier.

Installing Hook Functions

You install functions through the JAM 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 hook function type, one of the constant in Table 16. In this table, hook 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 16. Hook function types

Multi–function installation	Single-function installation	
SCREEN_FUNC	DFLT_SCREEN_FUNC	CKDIGIT_FUNC
FIELD_FUNC	DFLT_FIELD_FUNC	UINIT_FUNC
GRID_FUNC	DFLT_GROUP_FUNC	URESET_FUNC
GROUP_FUNC	KEYCHG_FUNC	RECORD_FUNC
PROTO_FUNC	ERROR_FUNC	PLAY_FUNC
TIMEOUT_FUNC	INSCRSR_FUNC	STAT_FUNC
CONTROL_FUNC	EXTERNAL_HELP_FUNC	VPROC_FUNC

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 hook 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 ( struct fnc_data );
```

Prototyped Functions

Prototyped functions are functions that get only the number and type of arguments that you specify. Prototyped functions are demand hook functions—that is, they must be invoked by name from a JAM component, such as a widget or screen.

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 hook calls a function, and JAM finds this function on the list of prototyped hook functions, JAM 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 JAM's 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 fields and groups do not get standard arguments, JAM has several library functions that let you get equivalent information about a field or group. `sm_inquire` can return a field's field 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 field calling a prototyped field function. Corresponds to the first standard argument to a field function.
SC_AFLDMDT	Bit mask that indicates the field's validation state and why the function was called. Corresponds to fourth standard argument of a field function.
SC_AFLDOCC	Occurrence number of the field that called the function. Corresponds to the third standard argument of a field 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 field function, a pointer to a copy of the field's contents, through `sm_getfield` or `sm_o_getfield`.

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 hook 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: JAM supports string and integer arguments, specified by `s` and `i`, respectively. The following declarations and definitions support the installation of two functions shown later in this chapter, `mark_flds` and `report`, and two JAM library functions. This code is usually found in the file `funclist.c`:

```
struct fnc_data proto_list[] = {
   SM_INTFNC ( ”mark_flds(i,i)”, mark_flds ),
   SM_INTFNC ( ”report(s,s)”, report ),
   SM_INTFNC ( ”sm_n_putfield(s,s)”, sm_n_putfield ),
   SM_INTFNC ( ”sm_gofield(i)”, sm_gofield )
};

int proto_count = sizeof ( proto_list ) /
                  sizeof ( struct fnc_data ) ;
```

In this example, `marks_flds` is prototyped to take two string arguments, `report` and `sm_n_putfield` take two string arguments, and `sm_gofield` takes a single integer argument. The last two functions are JAM library functions; their prototypes must conform to their definitions as shown in the *Language Reference*.

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`.

JAM supports any combination of strings and integers from zero to five arguments. JAM also supports 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, proto_list, &proto_count ) ;
```

Screen Functions

You can install an automatic screen function that JAM calls 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.

JAM executes the automatic screen function on both entry and exit for that screen. On entry, JAM executes the automatic screen function before it executes the screen's entry function. If a screen has a JPL module, JAM executes its unnamed procedure on screen entry before it executes the automatic function. On exit, JAM executes the screen's exit function before the automatic screen function.

JAM optionally recognizes overlay and reexposure of a screen as exit and entry events, respectively. This depends on how the JAM setup variable `EXPHIDE_OPTION` is set. If the variable is set to `ON_EXPHIDE`, JAM invokes screen exit and entry functions 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.

Note: *It is not advisable to open a screen from a screen entry function, since such an event yields undefined results.*

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_EXPOSE` and `K_ENTRY` 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)`

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 156.

```
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 JAM calls 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 widget's current state.

JAM executes the automatic field function on field entry, exit, and validation. You can install an automatic field function that JAM invokes 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 function access to non-standard information

Although automatic field functions cannot be prototyped, they can access non-standard information for specific fields through the field's memo edits. For an example, see page 161 later in this chapter.

Execution

JAM executes the automatic field function on all field events. On entry, JAM executes the automatic field function before it executes the field's entry function. On exit, JAM first calls the field's validation function, then its exit function, and finally the automatic field function. If the field has JPL validation, JAM executes this module after it executes the validation function.

Entry

JAM 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`.

Note: *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

JAM 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`.

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. Field 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 also trigger field validation unless the setup variable `IN_VALID` is changed from its default setting to `OK_NOVALID`.

- As part of screen validation. Screen validation occurs when the XMIT key is pressed. At that time, all fields on the screen are validated via the function `sm_s_val`. If the setup variable `XMIT_LAST` is set, screen validation also occurs when TAB or NL are pressed in the last field on a screen.

- When the application code calls library functions for field validation or screen validation.

If a field calls a validation function and belongs to a menu, radio button group, or checklist, JAM calls this function when the field is selected. JAM also calls the validation function of a checklist field when the field is deselected.

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 JAM 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.

Equivalent: `if(param4 & VALIDED)`

`MDT`
The field data changed since the current screen opened. Note that 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 data sets the `MDT`.

JAM never clears this flag. You can clear it with `sm_bitop`.

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_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.

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)

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 159.

```
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 Field 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 163.

```
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.

JAM 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, JAM first executes the grid’s entry function, 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.

JAM 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, JAM calls the grid row exit function before it calls the grid exit function.

Arguments

All grid functions receive three arguments in this order:

- An integer that contains the grid’s base field number—that is, the base field 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 grid function was called on grid entry.

Equivalent: `if(param3 & K_ENTRY)`

`K_EXIT`

The grid function was called on grid 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 grid 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 grid entry, exit, or validation. The intersection of this mask and the fourth parameter to the grid 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 in question. For grid 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.

Equivalent: `if((param3 & K_KEYS)==K_BACKTAB)`

`K_ARROW`
An arrow key caused the cursor to enter or exit the grid.

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. For grid 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_OTHER)`

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 ) ;
```

Group Functions

JAM calls group functions on entry, exit, and validation of groups. You can install an automatic group function that JAM invokes on entry, exit, and validation for all groups. Each group can invoke its own functions for these events through its entry, exit, and validation hooks. You can specify these hooks in the group's properties window, accessed through JAM's screen editor.

JAM executes the automatic group function on all group events. On entry, JAM executes the automatic group function before it executes the group's entry function. On exit, JAM first calls the group's exit and validation functions, and then calls the automatic group function. If the group has a JPL group module, JAM executes this module only after it executes the group hook functions.

JAM recognizes two events as group entry: when the cursor enters a group; and when the screen's current group is reactivated because an overlying window closes.

JAM recognizes two events 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 field that has its own validation function, JAM calls this function when the field is selected. JAM also calls the validation function of a checklist field when the field is deselected.

Note: *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.*

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 field. For a description of these flags, refer to page 126.

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.

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 164.

```
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 `ffuncs` as demand field functions:

```
sm_install ( GROUP_FUNC, gfuncs, &gcount ) ;
```

Help Function

The 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.

Arguments

The help function gets a single string that contains the help context identifier.

Returns

Returns either `PI_ERR_NONE` (success) or `PI_ERR_NO_MORE` (failure).

Installation

You can install only one function as the 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 166.

```
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 group function:

```
sm_install ( EXTERNAL_HELP_FUNC, &hlp_struct, (int *) 0 );
```

Timeout Functions

JAM 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. JAM calls each timeout function when its timeout interval elapses

Timeout functions are called from the lowest level of JAM 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 JAM 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. .

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.

Returns

A timeout functions should return a code that indicates whether JAM 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`, include a single timeout function `screen_saver` in the `fnc_data` structure `timeout_funcs`. To see the code for this function, refer to page 171. 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 ) ;
```

Key Change Function

A key change function can be called whenever JAM 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.

JAM calls the key change function once for each key that it gets from the keyboard or the playback hook function.

Note: *The key change function ignores any keys placed on the input queue by sm_ungetkey or jm_keys.*

Arguments

The key change function gets one integer argument, the JAM logical key that is read from the keyboard or received from the playback hook function.

Note: *The key change function is not called for the following keys: MNBR, ALSYS, and ALT keys.*

Returns

The key change function returns the key to be input into the application by sm_get_key. 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 171.

```
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

JAM calls the installed error function when it issues an error message—invoked either by a JAM error or by a call to one of JAM'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.

Arguments

The error function gets three arguments in this order:

- The number of the message to display—for JAM 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.

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 173.

```
sm_install ( ERROR_FUNC, &err_struct, (int *) 0 ) ;
```

Insert Toggle Function

JAM calls the insert toggle function when the data entry mode switches between insert and overstrike mode—for example, when the user chooses Insert. You can use this hook function to display a message that indicates the current mode.

JAM 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, JAM deinstalls its insert toggle function; the insert toggle function that you install can call JAM'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 174.

```
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

JAM calls the check digit function during validation of any field that is marked for check digit. A field is marked for check digit on its properties window, which you access through JAM's screen editor. You use a check digit function to perform your own check digit algorithm. If no check digit function is installed, JAM uses the library function `sm_ckdigit`, which 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 field contents.
- The occurrence number for the current field.
- The modulus as specified in the screen editor.
- The minimum number of digits as specified in the screen editor.

Returns

The check digit function should return 0 if the field passes check digit validation. If the function returns a non-zero value, JAM repositions the cursor to the offending field and the field 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

JAM calls the initialization and reset functions 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 JAM allocates its own memory structures or sets the physical display. Unlike other hook functions, the initialization function should be installed before `sm_initcrt` is called. Consequently, you cannot place the installation code for this hook function in the `funclist.c` function `sm_do_uinstalls`.

The reset function is called from the library function `sm_resetcrt` after JAM releases its memory and resets the physical display. Because JAM'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.

Note that 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 hook 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 hook 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 175.

Record and Playback Functions

JAM provides hooks for recording and playing back keystrokes. You can use this facility to create simple macros, or to perform regression testing on a JAM 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.

Note that 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. See the description of `sm_getkey` for more information.

Also note that 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 JAM 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 176.

```
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. JAM 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, JAM 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 179.

```
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

JAM 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 JAM library functions:

```
stat_text = sm_pinquire(SP_STATLINE);

stat_attr = sm_pinquire(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, JAM continues its usual processing and writes out the status line. If the function returns a non-zero value, JAM assumes that the hook 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 188.

```
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 JAM has no video file that supports a specific terminal type. JAM's output function calls the video processing function just before it displays data on a JAM screen; consequently, this function should perform only low-level processing.

Video processing functions should not call JAM 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 17.
- 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 17. For video processing codes that require no arguments, supply `NULL`.

Table 17. 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.

Code	Parameters	Action
V_COF		Turn cursor off.
V_CON		Turn cursor on.
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, JAM continues with normal processing. If it returns a non-zero value, JAM assumes that the hook 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 Error Functions

JAM's database drivers have three error 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 255 in the *Application Development Guide*.

Transaction Manager Hook Functions

The transaction manager builds a tree of all the 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. If any of these table views or server views has a hook function property specified, the transaction manager looks for it among the installed prototyped functions and calls it.

If the hook 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 model 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 hook functions, refer to page 384 in the *Application Development Guide*.

Arguments

The transaction manager hook 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 18 summarizes possible return codes for transaction manager hook functions:

Table 18. Return codes for transaction manager hook functions

Return value	Description
`TM_OK`	The event processing succeeded.
`TM_FAILURE`	The event processing failed.
`TM_PROCEED`	After completing the hook 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 database-based 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`	The event was not recognized.

Installation

You can specify a transaction manager hook function for each table view or server view in the screen. With the table view selected, enter the name of the hook function in the Database category's Function property. Note that if the hook function affects the SQL generation of `SELECT` statements, the hook function must be entered on the appropriate server view.

The hook 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 hook function cannot be found. As a result of this error, the transaction manager does not start its transaction.

Sample Hook Functions

The following sections show sample code for commonly used hook function types.

Prototyped

This section has two sample functions:

- `mark_flds` gets a range of values and highlights all fields whose data is within that range.
- `report` generates a report whose type and output device vary according to the supplied arguments.

Example 1

`mark_flds` gets a range of values and highlights all fields 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 fields on the screen with numeric values between the two arguments are temporarily highlighted. If the first argument is greater than the second, all fields on the screen with numeric values that are not between the two fields 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_err_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 field 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_err_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, " \n \nREPORT TYPE: %s \n", 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 JAM windows. */
   switch ( *report_type )
   {
   case 'F':
   case 'f':
      /* Output a field report */
      fprintf ( fp, " \nField Report: \n" ) ;

      /* Field Identifier and contents */
      cur_no = sm_getcurno ( );
      fprintf ( fp, "\tFIELD: %d (%s[%d]) = %s \n",
          cur_no,
          sm_name ( cur_no ),
          sm_occur_no ( ),
          sm_fptr ( cur_no ) ) ;

      /* Field sizes */
      size = sm_finquire ( cur_no, FD_LENG ) ;
      fprintf ( fp, "\tLENGTH: onscreen: %d "
          "Max: %d \n",
          size, sm_finquire ( cur_no,
          FD_SHLENG )
          + size ) ;

      fprintf ( fp, "\t# OCCURRENCES: onscreen: %d "
          "Max: %d \n",
```

```
            sm_finquire ( cur_no, FD_ASIZE ),
            sm_max_occur ( cur_no ) ) ;

        break;

    case 'S':
    case 's':
        /* Output screen report */
        fprintf ( fp, " \n \nScreen Report: \n" ) ;

        /* Screen Name */
        fprintf ( fp, "\tSCREEN: %s \n",
            sm_pinquire ( SP_NAME ) ) ;

        /* How much of screen is visible */
        fprintf ( fp, "\t%% VISIBLE IN VIEWPORT: %d \n",
            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, " \n \nWindow Stack Report: \n" ) ;

        /* Cycle through all the windows. */
        for ( select = 0 ;
            sm_wselect ( select ) == select ;
            select++ )
        {
            /* Window number... */
            fprintf ( fp, " \n\tWindow %d: \n",
                select ) ;

            /* Screen name */
            fprintf ( fp, "\t\tScreen: %s \n",
                sm_pinquire ( SP_NAME ) ) ;

            /* Number of fields and groups */
            fprintf ( fp, "\t\t# of Fields: %d "
                "# of Groups: %d \n",
                sm_inquire ( SC_NFLDS ),
                sm_inquire ( SC_NGRPS ) ) ;

            sm_wdeselect ( ) ;
        }
```

```
   sm_wdeselect ( ) ;

   break ;

case 'T':
case 't':
   /* Output display terminal report */
   fprintf ( fp, " \n \nTerminal Report: \n" ) ;

   /* Terminal Type */
   fprintf ( fp, "\tTERM TYPE: %s \n",
       sm_pinquire ( P_TERM ) ) ;

   /* Display mode */
   if ( sm_inquire ( I_NODISP ) )
      fprintf ( fp, "\tDISPLAY OFF \n" ) ;
   else
      fprintf ( fp, "\tDISPLAY ON \n" ) ;

   /* Input mode */
   if ( sm_inquire ( I_INSMODE ) )
      fprintf ( fp, "\tINSERT MODE \n" ) ;
   else
      fprintf ( fp, "\tTYPEOVER MODE \n" ) ;

   /* Block mode */
   if ( sm_inquire ( I_BLKFLGS ) )
      fprintf ( fp, "\tBLOCK MODE \n" ) ;

   /* Physical display size */
   fprintf ( fp, "\tDISPLAY SIZE: %d x %d \n",
       sm_inquire ( I_MXLINES ),
       sm_inquire ( I_MXCOLMS ) ) ;

   break;

default:
   /* Unrecognized report type */
   fprintf ( fp, "\tIllegal Report Type \n \n" ) ;
   return ( -3 ) ;
}

/* Once again, based on the type output... */
switch ( *report_out )
{
case '|' :
   /* It was a pipe, so close it. */
   pclose ( fp ) ;
   sm_err_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 = '\0';
      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_err_reset ( "Command Invoked" ) ;
   break ;

case '\0':
   /* 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] = '\0';

            /* Replace all space-newlines with %N */
            for ( ptr1 = ptr ;
                ptr1 = strchr ( ptr1, '\n' ) ;
                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_err_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 JAM associates 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 JAM calls hook 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_err_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_err_reset ( msg_buf ) ;

   return ( 0 ) ;
}
```

Automatic Field

This section has two sample functions:

- `auto_ffunc` puts general information about the current field on the status line.
- `memoval` uses a field’s memo edits to pass non-standard information to that field’s automatic field function.

Example 1

This function puts general information about the current field on the status line. This function is installed as the automatic field function in a JAM application; it is called on entry, exit, and validation for all fields.

On field entry, the function places information about the field on the status line:its name, if any, number, and occurrence offset. If the field is a selected member of a group—for example, radio buttons or checklists—the status line shows the text of the selected field, 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 field function shows how to use a field's memo edits to pass non-standard information to that function. This function validates each field against the contents of its memo edits.

This function is to be installed as a non-prototyped field validation function in a JAM application, either on the `FIELD_FUNC` list or as the `DFLT_FIELD_FUNC`.

The function validates fields 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 Field

The following local field hook functions can be called by individual fields to perform initialization and validation based on external criteria:

Two field functions to include on the field function list are defined here. The first one, `fentry`, initializes the value in a field provided that it has not changed since the screen was opened. The second one, `fvalid`, validates the contents of a field. 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 fields by pressing the X key. On group exit, `auto_gfunc` deinstalls `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 171.

```
/* 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 ;   /* Hook 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 JAM uses to invoke its own help facility, created with Dynatext™. 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 JAM; 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 Hook 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 JAM executive is in use. If the JAM 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 JAM 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 JAM_EXECUTIVE
      * macro is not defined in any JAM header file. It
      * is used here to distinguish between applications that
      * use the JAM executive and those that don’t.
      */
      if (
#ifdef JAM_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 hook 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'\n", 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` hook function, called whenever JAM 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 JAM overwrites the status line with messages, thus destroying the INS/OVR message.

The last column of the status line is not written; JAM 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 hook. 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 = ’\0’ ;
```

```
   term[ 29 ] = '\0' ;
   if ( ptr = strchr ( term , '\n' ) ) * ptr = '\0' ;

   /* 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.\n", hours, minutes, seconds ) ;

   return ( 0 ) ;
}
```

Record and Playback

The following example shows how record and playback might work together in a JAM regression test.

The two functions `record` and `play` implement a simple mechanism for recording and later playing back keystrokes in a JAM 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_err_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_err_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 fields on the current screen with numeric values less than zero with an attribute change. The function `mark_high` marks all fields on the current screen with numeric values higher than 1000. The same functionality is duplicated as the prototyped function `mark_flds` (see page 148).

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 ( control_string )
char *control_string;/* Control string text passed by JAM */
{
   /* Mark all fields less than zero */
   return ( mark_flds ( 0, MARK_LT ) ) ;
}

int
mark_high ( control_string )
char *control_string ;/* Control string text passed by JAM */
{
   /* 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 field, 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 JAM message window.

`report` is installed under four names in the `CONTROL_FUNC` function list for JAM 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: field, 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 pop-up JAM message window. For example, the following control string will generate a report about the current field and display the report in a pop-up 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 = '\0';

      /* 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_err_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 JAM windows. */
if ( ! strcmp ( report_type, ”rep_field” ) )
{
   /* Output a field report */
   fprintf ( fp, ” \n \nField Report: \n” ) ;

   /* Field Identifier and contents */
   fprintf ( fp, ”\tFIELD: %d (%s[%d]) = %s \n”,
       cur_no = sm_getcurno ( ),
       sm_name ( cur_no ),
       sm_occur_no ( ),
       sm_fptr ( cur_no ) ) ;

   /* Field sizes */
   fprintf ( fp, ”\tLENGTH: onscreen: %d ”
       ”Max: %d \n”,
       size = sm_finquire ( cur_no, FD_LENG ),
       sm_finquire ( cur_no, FD_SHLENG )
       + size ) ;

   fprintf ( fp, ”\t# OCCURRENCES: onscreen: %d ”
       ”Max: %d \n”,
       sm_finquire ( cur_no, FD_ASIZE ),
```

```
        sm_max_occur ( cur_no ) ) ;
}
else if ( ! strcmp ( report_type, "rep_screen" ) )
{
   /* Output screen report */
   fprintf ( fp, " \n \nScreen Report: \n" ) ;

   /* Screen Name */
   fprintf ( fp, "\tSCREEN: %s \n",
       sm_pinquire ( SP_NAME ) ) ;

   /* How much of screen is visible */
   fprintf ( fp, "\t%% VISIBLE IN VIEWPORT: %d \n",
       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, " \n \nWindow Stack Report: \n" ) ;

   /* Cycle through all the windows. */
   for ( select = 0 ;
       sm_wselect ( select ) == select ;
       select++ )
   {
      /* Window number... */
      fprintf ( fp, " \n\tWindow %d: \n",
          select ) ;

      /* Screen name */
      fprintf ( fp, "\t\tscreen: %s \n",
          sm_pinquire ( SP_NAME ) ) ;

      /* Number of fields and groups */
      fprintf ( fp, "\t\t# of Fields: %d "
          "# of Groups: %d \n",
          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, ” \n \nTerminal Report: \n” ) ;

   /* Terminal Type */
   fprintf ( fp, ”\tTERM TYPE: %s \n”,
       sm_pinquire ( P_TERM ) ) ;

   /* Display mode */
   if ( sm_inquire ( I_NODISP ) )
      fprintf ( fp, ”\tDISPLAY OFF \n” ) ;
   else
      fprintf ( fp, ”\tDISPLAY ON \n” ) ;

   /* Input mode */
   if ( sm_inquire ( I_INSMODE ) )
      fprintf ( fp, ”\tINSERT MODE \n” ) ;
   else
      fprintf ( fp, ”\tTYPEOVER MODE \n” ) ;

   /* Block mode */
   if ( sm_inquire ( I_BLKFLGS ) )
      fprintf ( fp, ”\tBLOCK MODE \n” ) ;

   /* Physical display size */
   fprintf ( fp, ”\tDISPLAY SIZE: %d x %d \n”,
       sm_inquire ( I_MXLINES ),
       sm_inquire ( I_MXCOLMS ) ) ;

}
else
{
   /* Unrecognized report type */
   sprintf ( msg_buf, ”Illegal report type %s”,
       report_type ) ;
   sm_err_reset ( msg_buf ) ;
   fprintf ( fp, ”%s \n \n”, 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_err_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 = '\0';
         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_err_reset ( "Command Invoked" ) ;
      break ;

   case '\0':
      /* 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] = '\0';
```

```
      /* Replace all space–newlines with %N */
      for ( ptr1 = ptr ;
          ptr1 = strchr ( ptr1, '\n' ) ;
          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_err_reset ( "File appended" ) ;
      break ;
   }
   return ( 0 ) ;
}
```

Status Line

The following example shows how to write a status line hook 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 hook 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 back into JAM 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 ) ;
}
```

Moving Data Between Screens

JAM lets you move data between screens two ways:

- Load and activate screens as *local data blocks*, or LDBs. LDBs automatically initialize and capture field data on screen entry and exit, respectively.
- Issue JPL `send` and `receive` commands or their equivalent library functions. These let you explicitly write and read screen data to and from temporary buffers.

Using Local Data Blocks

JAM 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, JAM uses its fields, or *LDB entries*, to transfer data to and from corresponding fields on the current screen. By using LDBs, applications can transfer data between screens automatically.

JAM matches LDB entries and screen fields by name. Only named fields 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 JAM enters or exits a screen, it

checks whether any LDBs are active. If one or more LDBs are active, JAM performs LDB write-through as follows:

- At screen entry, JAM initializes or overwrites fields 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, JAM writes LDB values to the screen after it executes the screen's entry function.

  The LDB always overwrites existing screen data, even if the field has input protection. One exception applies: at screen open, the LDB respects initial field data specified through the screen editor.

- At screen exit, JAM writes field 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 of `EXPHIDE_OPTION`. JAM writes screen data to the LDB before it executes the screen's exit function.

If data is transferred between arrays, JAM allocates for the target array the number of occurrences required to accommodate the incoming data, up to the array's maximum number of occurrences.

LDB write-through versus 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 fields if the LDB also writes to those fields. However, you can circumvent this restriction as follows:

1. Write a procedure or function that populates the fields with the desired values.
2. Attach this procedure or function to an unused logical key—for example, `APP22=^myproc`.
3. 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, JAM's screen manager pops all data off the input queue. Given the previous example, when JAM gets `APP22`, it calls `myproc` and executes its contents.

selection groups

JAM regards the selections that the user make in a selection group—radio buttons, toggle buttons, checkboxes, 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 fields arranged in the same order, and have the same contents.

restrictions

JAM 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: static labels, lines, and boxes.

LDB entries and their corresponding fields 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 field is shorter than the source data, JAM truncates the data.
- If the maximum number of occurrences specified for the target LDB entry or field is less than the number of occurrences allocated for the source, JAM discards the overflow occurrences.

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 active LDBs have entries with the same name, JAM uses the entry on the most recently activated LDB.

LDB handles

JAM 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 JAM 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 field- and screen-level processing. If you display LDB screens, be careful to safeguard this transitional data.

Default Activation

At application startup, JAM 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.jam | screen2.jam | screen3.jam
   ```

3. Looks for the library `ldb.lib` and the screens stored in it.
4. If `ldb.lib` does not exist, JAM searches the path for screen `ldb.jam`.

Runtime Loading and Activation

JAM 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 field data.

push and pop LDBs

JAM 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. JAM 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 LDB's 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

JAM provides several functions that let you get information about loaded and active LDBs and manipulate their values:

- `sm_ldb_get_active`, `sm_ldb_get_next_active`—tell which LDBs are active and their order of activation.
- `sm_ldb_get_inactive` and `sm_ldb_get_next_inactive` let you determine which LDBs are loaded but inactive and in what order they were 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.

field references and LDB entries

Library functions and JPL procedures that reference fields 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, JAM reverses the search order: on screen entry or exit, JAM reverses the search order and first looks in the LDB for the requested data. JAM writes LDB data to the screen after screen entry and writes it 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, JAM looks for a corresponding field 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*

Sending and Receiving Data

JAM 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 field-level hook—entry, exit, and validation—to send and receive data. As developer, you explicitly tell JAM what data to send and when to send it. You also avoid unintentionally overwriting field 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 field length decides whether incoming data is received whole or is truncated.
- Send/receive operations do not require the source and target fields 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 the *Language Reference*.

Bundles

Send and receive commands and functions act on bundles, which provide temporary storage for the data you wish to transfer between screens. You can name bundles for explicit access. JAM can maintain up to ten bundles, including one that can be unnamed. If you send data without specifying a bundle name, JAM writes the data to an unnamed bundle; this data is available to the next receive request that omits specifying a bundle name.

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 field 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 JAM 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 !(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 fields. For example, the following `receive` command reads `bundle1` and puts its data into three fields:

```
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. JAM does not check whether receive data is valid for the target fields.

Unless the `receive` command includes the `keep` argument, when it returns, JAM 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 JAM 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

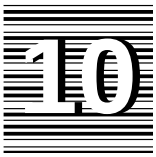
bundle and the number of occurrences in each item. This example also gets the data from each specified item occurrence through successive calls to `sm_get_bundle_data`.

```
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`.

JAM also provides these library functions:

- `sm_get_next_bundle` 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.



Error Handling and Messages

JAM supports plain messages, status messages, and dialog messages. Messages can be sent to the status line or a window can be created to display a message. Messages can be hard coded into the application or stored in a message file. Message files can be loaded at initialization or at any time. JAM allows you to translate certain message constants—these are found in the provided message file. For more information on the message file, refer to page 41 in the *Configuration Guide*. You can also write a hook function that executes every time one of the error message display functions is called. The error hook function is described below. Described at the end of this chapter is the status line display hierarchy.

Table 19 describes the functions which display errors, messages, and status information, as well other functions related to message storage and retrieval. For more detail see the individual reference pages for these functions in the *Language Reference*.

Table 19. Error and message related functions

Function	Description	Comments
sm_d_msg_line	Display message in status line	Can change display attributes of message.
sm_femsg	Display message in status line or window	Awaits user acknowledgement. (Messages are always displayed in a window in GUI environments. Character JAM chooses message placement based on message length. Use MESSAGE_WINDOW set-up variable to configure character JAM behavior.)
sm_ferr_reset	Display message in status line or window	Identical to sm_femsg when displayed in window. When displayed on status line, puts cursor on at current field.
sm_fqui_msg	Display message in status line or window	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	Display message in status line or window	Identical to f_err_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	Get initialization error message	For example, if a call to sm_msginit is unable to initialize a message file—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.
sm_m_flush	Flush the status line	Forces JAM to display updates to the status line. This is useful if you want to display the status of an operation with sm_d_msg_line without flushing the entire display (e.g. with sm_flush).
sm_message_box	Display dialog option box	Creates a dialog box that displays a message and requests the user to select a button. (e.g. Yes/No, Abort/Retry/Cancel). JAM prevents further interaction with the application until the function returns with the user's selection.
sm_msg	Display message at specific column of status line	Merges the specified message with the current contents of the status line and displays it at the specified column.

Function	Description	Comments
sm_msg_get	Get contents of stored message	Gets a message from a message file previously loaded by sm_msgread. Message files are binary files, created through the JAM utility msg2bin. If not found, return message class and number.
sm_msg_find	Get contents of stored message	Finds the message specified and returns a string. Unlike sm_msg_get this function returns 0 if the message is not found.
sm_msgread	Read/load/delete messages	Reads a single set of messages from a binary message file into memory, based on class and prefix values. access these loaded messages through sm_msgget or sm_msg_find You can also use sm_msgread to delete messages, or to read them but not load them into memory so they can be fetched from the disk when requested, saving substantial memory.
sm_setbkstat	Display low-priority message in status line	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.

Note: *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.*

Error Hook Function

JAM calls the error function whenever you call one of JAM's error message display routines—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. Refer to page 137 for more information.

Status Line

JAM reserves one line on the display for error and other status messages. Many terminals have a special status line. If not, JAM uses the bottom line of the display for messages.

There are several types of messages that use the status line; they are described here in order of their priority from highest to lowest:

- Error or any other overwrite message
- Status text from `sm_d_msg_line`
- “Wait” message
- Field and menu item status text
- Form status text
- “Ready” message
- Background status text

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`. All these functions wait for the message to be acknowledged. Their messages 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 is 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 JAM message file.

Field/Menu item status

When the status line has no higher priority text, the screen manager checks the current field or selected menu item for text to be displayed on the status line. If the

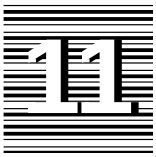
cursor is not in a field or on a menu item, or if the current field or item has no status text, JAM looks for background status text.

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.

SECTION THREE

The SQL Executor



Database Initialization

Before your application can access any data, you first have to connect to an initialized database engine. This chapter describes how to initialize one or more database engines. Chapter 12 describes how to connect to an engine.

When you initialize database engines, you are setting which database engines will be available for your application. After an engine is initialized, you can connect to it using the transaction manager, JPL procedures, or C functions.

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 *JAM database driver* is a C library or a DLL that handles all communication between JAM and a DBMS. A *JAM support routine* is the name of the main entry point, or function, in a JAM database driver. Database initialization tells JAM which database driver and which support routine to use to access a DBMS. JAM supports static initialization on all platforms and dynamic initialization on a subset of platforms.

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

In dynamic initialization, the application identifies the desired support routines at runtime. No compilation is needed to change the initialization. For example, in Windows, the `JAM7.INI` file supports dynamic initialization.

By default, JAM is distributed with a driver for JYACC's database engine, JDB. Drivers are also available for other database engines.

In a JAM application, you can access one or more database engines. The application must have a driver for each engine, and it must initialize the engine before declaring a connection.

Using dbiinit.c for Static Initialization

A list of the support routines available for your application is included in the module `dbiinit.c`. This file is automatically created from settings found in your makefile when you first build your `jamdev` and `jam` executables. When you run JAM, the library function `dm_init` is called for each support routine listed in `dbiinit.c`.

If the initialization is successful, the support routine returns zero. In some cases, the support routine rejects the initialization and returns an error code. In these cases, there might be 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 `jmain` or `jxmain`, JAM will display an error message and terminate.

If necessary, you can create a new version of `dbiinit.c`. First, you need to delete the current `dbiinit.c` file, edit the database settings in the JAM makefile, and then run the makefile. For more information, refer to page 211.

Description of dbiinit.c

The file `dbiinit.c` contains:

- A function declaration for one or more support routines and a corresponding transaction model.
- 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[] =
{
/* {"engine-name", support-routine, case-flag , (char *) 0}, */
   {"jdb", dm_jdbsup, DM_DEFAULT_CASE, (char *) 0},
   { 0, 0, 0, 0 }
};
```

engine-name can contain any character string you wish. If an application uses two or more database engines, the mnemonic *engine-name* tells JAM which database engine to use. Most of the examples in this manual use a vendor name as the mnemonic, for example `sybase` or `oracle`, but any character string that is not a keyword is valid. Engine names are case-sensitive.

The *support-routine* name is usually in the form `dm_`*vendor_code*`sup` where *vendor_code* is an abbreviated vendor name. Some examples are:

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

case-flag determines how JAM uses case when mapping column names to JAM variables for SQL `SELECT` statements. JAM variables can be widgets on the screen, JPL variables, or LDB variables. Using the case setting, you can create all your JAM variables in a particular case and have JAM do the conversion to that case for you. The *case-flag* values are described in Table 20.

Table 20. case-flag Options

case_flag	Option	Description
`DM_PRESERVE_CASE`	`p(reserve)`	Use the case returned by the engine. The column names must exactly match the JAM variable names.
`DM_FORCE_TO_LOWER_CASE`	`l(ower)`	Force all column names returned by an engine to lower case. For this value, use lower case when naming JAM variables.
`DM_FORCE_TO_UPPER_CASE`	`u(pper)`	Force all column names returned by an engine to upper case. For this value, use upper case when naming JAM variables.
`DM_DEFAULT_CASE`	`d(efault)`	Use the default value set by JYACC in the support routine. Refer to the Database Drivers section in the *Database Guide* to find the value for a specific engine.

For example, ORACLE returns all column names in upper case. If the case flag is set to `DM_PRESERVE_CASE`, the application needs JAM variables with upper case

names. To map columns to JAM 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 JAM variables, you can set the case flag to either DM_FORCE_TO_UPPER_CASE or DM_FORCE_TO_LOWER_CASE.

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

Using JAM7.INI for Dynamic Initialization

In JAM for Windows, you have two methods of initializing a database engine. One method, which is described in the previous sections, compiles support for a database into the executable. The other method uses specifications in your JAM7.INI file to initialize database engines at runtime when the JAM program is started.

To set the database engines in your JAM7.INI file, add a database-specific section to the file. The syntax is:

```
[databases]
installed = engine-name  [engine-name]

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

engine-name can contain any character string you wish. If an application uses two or more database engines, the mnemonic *engine-name* tells JAM which database engine to use. Most of the examples in this manual use a vendor name as the mnemonic, for example sybase or oracle, but any character string that is not a keyword is valid. Engine names are case-sensitive. This entry is required.

case determines how JAM uses case when mapping column names to JAM variables for SQL SELECT statements. JAM variables can be widgets on the screen, JPL variables, or LDB variables. Using the case setting, you can create all your JAM variables in a particular case and have JAM do the conversion to that case for you. The case options are described in Table 20. This entry is optional.

driver DLL is provided by JYACC for use with a particular database engine and is set by the installation program. For additional information about the DLL for a specific engine, refer to the README.*vendor-code* file in the \JAM7\NOTES directory. (*vendor-code* is a three letter abbreviation for the vendor, for example, syb for SYBASE.) This entry is required.

model DLL is the name of the transaction model provided by JYACC for use with the transaction manager. For information about the transaction model for a specific

engine, refer to the `README.`*vendor-code* file in the `\JAM7\NOTES` directory. This entry is required if you are using transaction manager.

Initialization Procedure

As a part of initialization, JAM calls the support routine for information on the particular DBMS. This includes the following information:

- The engine's capabilities (e.g., whether the engine can execute stored procedures or support multiple connections).
- The required formatting for character, date, and null strings being passed to the database.
- The default for case handling.

In addition, JAM 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` *engine-name*

or sets the *current engine* for a statement by including the `WITH ENGINE` clause. If an application initialized 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.

Making a New dbiinit.c to Change Static Initialization

When you run the JAM makefile, the `mkinit` command creates a new version of `dbiinit.c`, if one does not already exist. To make a new version of `dbiinit.c`, delete the current version, edit the database engine settings in the JAM makefile, and run the makefile containing the new settings. The command appears in the makefile as follows:

`mkinit` *[*`–dnlup`*]dbs=engine-name ...*

The square brackets indicate the optional command flags and do not need to be typed. You need to enter one of the command flags for each engine.

Options and Arguments

Most of the command flags deal with the case conversion for column names. When a query retrieves a column value, JAM looks for a JAM 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 JAM variables in a particular case and have JAM do the conversion to that case for you.

`-d` Use the default case conversion set in the support routine. Refer to the Database Drivers section in the *Database Guide* to find the setting for a particular database engine.

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

`-l` Convert the column names to lower case JAM variable names.

`-u` Convert the column names to upper case JAM variable names.

`-p` Preserve the case of the column names when converting to JAM variable names.

dbs

A three-letter abbreviation that JAM specifies for the database. For example, the abbreviation for JDB is `jdb`. The abbreviation for SYBASE is `syb`. Refer to the Database Drivers section in the *Database Guide* to find the abbreviation for a particular database engine.

If you specify an invalid abbreviation, you will get an unresolved external error when you try to link.

engine-name

The name you want to use for the engine in `DBMS ENGINE` statements and `WITH ENGINE` clauses.

Database Connections

Once the engine is initialized, you need to establish a connection before your application can access any data. You can use the Connection option on the Database menu or you can use the `DBMS DECLARE CONNECTION` command in a JPL procedure or C function.

Connecting to a Database Server

Once a database engine is initialized, the application must connect to it before it can perform operations. The `DBMS` command for declaring a connection is:

```
DBMS [WITH ENGINE engine-name] \
    DECLARE connection-name CONNECTION FOR \
    OPTION "argument" [OPTION "argument"]
```

If a `WITH ENGINE` clause is not specified, the connection is declared for the default engine. The connection names specified in the statement are case-sensitive.

Different engines support different options. Common options include `USER`, `PASSWORD`, `DATABASE` and `SERVER`. If an option included in the `DECLARE CONNECTION` statement is not supported by the engine, the database driver reports error number 53254—`Bad arguments`. To see a list of options for a specific engine, refer to the Database Drivers section in the *Database Guide*.

Alternatively, in JAM editor programs like `jamdev`, there is a Connection option on the Database menu. After you choose a database engine, the options available for that engine are displayed.

Description of a Database Connection

A declared connection is a named structure describing a session about an engine. This information includes:

- A connection name.
- Logon information supplied by the option arguments, for example, a user and database name.
- 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, JAM is able to close all open cursors on the connection).

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

Setting the Default Connection

When you are using multiple connections, you should set a default connection. This is done with the following command:

`DBMS CONNECTION` *connection-name*

You can override the default connection using a `WITH CONNECTION` clause which specifies a connection to be used for a single statement. For example:

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

Remember that a connection is always associated with an initialized engine. Setting a connection as the current or default connection also sets the current or default engine.

Connections to Multiple Engines

If an application is using two or more engines, a connection must be declared for each engine. You can then set a default connection. For example:

```
DBMS WITH ENGINE sybase DECLARE sybcon CONNECTION FOR \
   USER ":uname" PASSWORD ":pword" SERVER "birch"
DBMS WITH ENGINE oracle DECLARE oracon CONNECTION FOR \
   USER ":uname" PASSWORD ":pword"
DBMS CONNECTION sybcon
DBMS SQL SELECT * FROM titles WHERE title_id = :+title_id
```

In the example, connections are declared on the engine `sybase` and the engine `oracle`. JAM will get the values for `USER` and `PASSWORD` from the variables `uname` and `pword` at runtime. The connection `sybcon` is chosen as the default. Therefore, JAM performs the `SELECT` on the connection `sybcon` and uses the support routine of `sybcon`'s engine to execute the query.

Multiple Connections to a Single Engine

Some database engines permit two or more simultaneous connections. Refer to the Database Drivers section in the *Database Guide* to see if this option is available for a specific engine. If you wish to take advantage of this feature on a valid engine, you need to declare a named connection for each session on the engine. For example:

```
DBMS ENGINE sybase
DBMS DECLARE s1 CONNECTION FOR \
   USER ″:uname″ PASSWORD ″:pword″ SERVER ″birch″
DBMS DECLARE s2 CONNECTION FOR \
   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`. JAM will get 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 JAM 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, JAM returns the error `DM_ALREADY_ON`.

Closing Connections

The application can close connections by executing the following command for each declared connection:

`DBMS CLOSE CONNECTION` *connection-name*

Alternatively, it can close all connections on an engine by executing the following command:

`DBMS` *[*`WITH ENGINE` *engine-name]* `CLOSE_ALL_CONNECTIONS`

The Disconnect option on the Database menu uses this command to close the connections on a specified engine.

Using Cursors

A *cursor* is a SQL object associated with a specific query or operation. JAM 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 (e.g., the number of rows to fetch, information on target variables or binding parameters, etc.).

When a cursor is declared, JAM 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. JAM frees the structure when the cursor is closed. Cursor names are case-sensitive so `CUR1` and `cur1` are two distinct names.

Every connection has one or two default cursors which JAM automatically creates. An application may also declare named cursors on a connection. A JAM application may use either or both of these types of cursors.

The default cursors are convenient for SQL statements that are executed once, and for applications using only one select set at a time. All database commands executed with the JPL command `DBMS SQL` use default cursors.

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.

The rest of this chapter describes the use of cursors in an application. Please note that the discussion of how data is passed between an application and a database is covered in Chapter 15.

JAM provides several `DBMS` commands 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`. They are discussed in Chapter 14 of this manual and in Chapter 11 of the *Database Guide*.

Using a Default Cursor

For most engines, JAM 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, JAM 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 the Database Drivers section of the *Database Guide*.

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 SQL WITH CONNECTION c1 \
   SELECT title_id, name FROM titles \
   WHERE genre_code = 'ADV'
DBMS SQL 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, JAM uses the default connection `c2`.

An application may also close the default cursor if it is not needed. For more information, refer to page 222.

Using a Named Cursor

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(s).
- Close cursor(s).

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, JAM uses the default connection.

```
DBMS [WITH CONNECTION connection-name ] \
    DECLARE cursor-name CURSOR FOR SQL-statement
```

An application may 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 may be executed any number of times. The name of the cursor must be a valid JAM identifier. The cursor name is case-sensitive.

Supplying Values Using Colon Expansion

A cursor may 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 de-referenced when the cursor is declared. It is *not* de-referenced when the cursor is executed. An application may use colon variables or colon-plus variables anywhere in the statement.

Supplying Values Using Binding

To de-reference 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. Note that 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 JAM expression for each bind tag. When the example is executed, the application will fetch all rentals where `rental_date` is the value of the variable `today`. To execute the select again where `rental_date` is another value, change the contents of `today` and re-execute the cursor:

```
today = @date(today) – 1
DBMS WITH CURSOR c1 EXECUTE USING rental_date = today
```

You can also supply a new variable for the bind tag:

```
DBMS WITH CURSOR c1 EXECUTE USING rental_date = yesterday
```

Literals and expressions are also 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 JAM concatenation operator to append the contents of the `title_val` variable followed by `%` 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, JAM 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
```

JAM 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

Note that 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 SQL UPDATE ....
```

When cursor `cur1` is declared JAM 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, the JAM performs the `SELECT` on connection `sybcon`. The default connection `oracon` performs the subsequent `UPDATE`.

Using a Transaction Manager Cursor

Even though a cursor is usually declared with a SQL statement, a different syntax is available for use in the transaction manager. 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 hook function to declare the cursor and to execute any additional processing. Then, subsequent transaction events attach the SQL statement by re-declaring the cursor.

In the following example, the `make_cursor` hook 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 hook function is called and is followed by the transaction events that re-declare and

execute the cursor with the applicable SQL statement, writing the select results to the `title_all` variable.

```
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
```

Modifying a Cursor

A cursor may be re-declared 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
```

You can also modify the cursor behavior, if the cursor is associated with a SQL `SELECT` statement, using additional `DBMS` commands provided by JAM. These commands include `ALIAS`, `CATQUERY` which can be used with `FORMAT`, `COLUMN_NAMES`, `OCCUR`, `START`, and `UNIQUE`. They are discussed in Chapter 11 of the *Database Guide*. Here we note that these settings are not lost when a cursor is re-declared, but only when the cursor is closed.

A cursor cannot be re-declared for a different connection. To re-use the cursor name on a different connection, you must first close the cursor.

Closing a Cursor

To close a cursor and free its data structure, execute the following:

`DBMS CLOSE CURSOR` *cursor-name*

Cursors are also closed when the application closes the the connection.

To close the default cursor, execute:

```
DBMS CLOSE CURSOR
```

The default cursor remains closed unless the application executes a `DBMS SQL SELECT` statement. JAM will automatically re-open the cursor when it is needed.

Reading Information from the Database

The SQL executor provides access to a database engine through one of JAM's database drivers. You can enter SQL statements using the SQL syntax supported by your database engine. In the SQL executor, you also have access to a series of commands to help you return the information to JAM variables. These commands are included in each of JAM's database drivers and can be used in either JPL procedures or C functions.

A JAM 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 JAM application, in particular the destination and format of data returned by SQL `SELECT` statements. For more information about error and status codes, refer to page 255.

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 section of the *Database Guide* for more information.

The information on how data is mapped to JAM variables also applies to processing in the transaction manager even though most of the examples in this chapter use JPL procedures and colon processing to construct the SQL `SELECT` statements.

Fetching Data Using SELECT Statements

When a `SELECT` statement is passed to an engine, JAM performs several steps before transferring data to JAM variables.

1. JAM 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 JAM variable destination. If a destination exists, JAM records the length of the variable. If no JAM destination exists for a column, or if the destination is an LDB variable with initialized text, JAM does no fetches for the column. Refer to the following section for more information on JAM destinations.
3. It determines the number of rows to fetch. This number usually equals the number of occurrences in the smallest JAM destination variable, or 0 if there are no target variables. Refer to page 228 for more information.
4. Finally, JAM 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 edit. Refer to page 234 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 JAM variable. Refer to page 237 for more information.

JAM 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 JAM destination. There are two ways of associating JAM target variables with database columns.

- Give a JAM target variable the same name as a database column. This is called *automatic mapping*.
- Explicitly declare a JAM variable as the target of a database column. This is called *aliasing*.

Automatic Mapping

By default when executing a `SELECT` statement, JAM will search for variables with the same names as the specified columns. These JAM variables can be widgets, JPL variables, or LDB variables. For the statement,

```
DBMS SQL SELECT title_id, name, pricecat FROM titles
```

to return values to JAM 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 JAM 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 JAM variable for each column. The variables can be named `title_id`, `name` and `pricecat`. If one of these variables does not exist, JAM ignores the values belonging to that particular column.

JAM also permits the use of the `*` in the `SELECT` statement,

```
DBMS SQL SELECT * FROM titles
```

Using automatic mapping, JAM 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 SQL SELECT titles.title_id, titles.name,
   titles.pricecat FROM titles
```

The JAM 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 JAM 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 JAM variables for a `SELECT` statement should have lower case names. If the case flag is `DM_FORCE_TO_UPPER_CASE`, the JAM variables should have upper case names. If the case flag is `DM_PRESERVE_CASE`, the JAM variables should match the exact case of the database columns. For information on a particular engine's case flag, refer to the Database Drivers section of the *Database Guide*.

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 JAM variable name.
- A column whose name conflicts with other JAM 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.

JAM 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 JAM variable:

```
DBMS [WITH CURSOR cursor-name ] ALIAS column1 jam-var1 \
    [, column2 jam-var2 ... ]
```

The following syntax aliases a column position to a JAM variable:

```
DBMS [WITH CURSOR cursor-name ] ALIAS jam-var1 \
    [, jam-var2 ... ]
```

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 *jam-var* should always match the exact case of the JAM variable name. For information on a particular engine's case flag, refer to the Database Drivers section of the *Database Guide*.

If an application aliases a column to a JAM variable that does not exist, JAM 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 JAM variables. For example,

```
DBMS ALIAS first_name first, last_name last
DBMS SQL SELECT cust_id, first_name, last_name FROM customers
```

JAM 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 11.

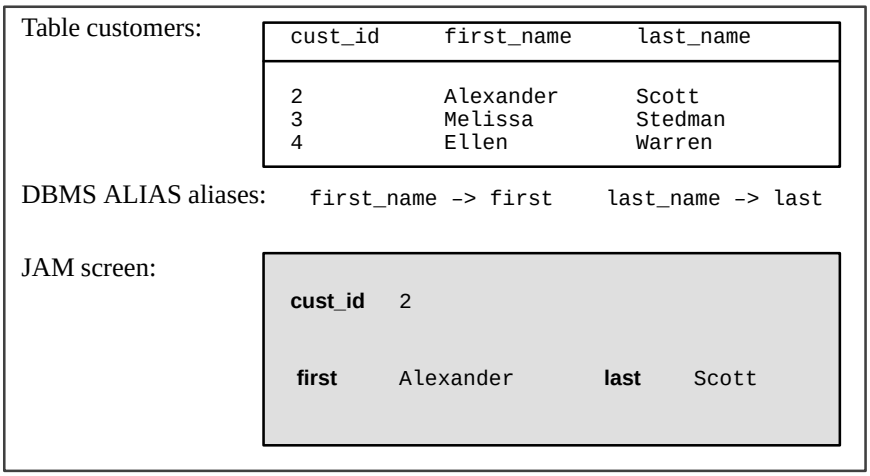


Figure 11. The mapping of a SELECT statement when aliases are used.

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 JAM 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 JAM identifier, enclose it in quote characters; this ensures that JAM parses it correctly.

Aliasing by Column Positions

Now consider an example that uses positional aliases. For example,

```
DBMS ALIAS min_rent, max_rent, avg_rent
DBMS SQL SELECT MIN(num_rentals), MAX(num_rentals),
   AVG(num_rentals) FROM customers
```

JAM 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`. Note that 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 JAM 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, JAM will attempt to write the values of the first three columns to the three positional alias variables. If those variables are no longer available, JAM 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*. Please note that 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, JAM will also support it. You can follow the syntax of the engine, replacing *heading* with the name of the appropriate JAM variable.

For example, if the syntax shown above is supported by the engine, than the following could be used in a JAM application,

```
DBMS SQL SELECT title_id id, name, pricecat price FROM titles
```

When this statement is executed, the DBMS tells JAM that the columns `id`, `name`, and `price` were selected. JAM will look 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. JAM provides identical support for `DBMS ALIAS` on all engines.

Fetching Multiple Rows

A select set often contains more than one row. JAM must determine how many rows it can fetch at one time from a select set. Subsequent rows in the select set are fetched by executing one or more `DBMS CONTINUE` commands.

Determining the Number of Occurrences

JAM 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 Yes, only one row is fetched.
- If using browse mode, only one row is fetched. (Refer to the Database Drivers section of the *Database Guide* to see if the engine supports browse mode.)
- Otherwise, JAM 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, JAM fetches a row for each occurrence. If the targets do not have the same number of occurrences, JAM 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 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 SQL 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 at a time.

Also, consider an application that executes a `SELECT` in a screen entry function. By default, JAM first searches the LDB and then the screen for JAM 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 JAM 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 Yes. Set the # of Occurrences property or leave it blank, which indicates an unlimited number of occurrences. Because the application must keep the entire select set in memory, the realistic limit might be much lower on a platform like MS-DOS 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; JAM allocates only whatever memory is needed to hold the returned rows. Therefore, a JAM screen might contain variables each with 10 elements and 1000 occurrences. If a select set contained only 75 rows, JAM 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. JAM 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 page 255 for more information on @dmretcode and related variables. An example procedure is shown below:

```
proc select_all
DBMS SQL 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. JAM 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 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.

Note: *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 SQL 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_UP	scrolls up a screenful of rows
DBMS CONTINUE_TOP	scrolls to the first screenful of rows
DBMS CONTINUE_BOTTOM	scrolls to the last 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, JAM 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, JAM buffers SELECT rows in a temporary binary file. The syntax of the command is:

DBMS *[* WITH CURSOR *cursor-name]* 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 SQL 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 JAM 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 JAM 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 131 in the *Database 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 JAM to read and discard *int* – 1 rows before writing the rest of the select set to JAM variables.

Refer to Chapter 11 in the *Database Guide* for more information on this command.

Format of Select Results

Before writing a database column value to a JAM variable occurrence, JAM determines the data type of the database column.

In all cases, if the value equals the engine's null (e.g., NULL), JAM clears the variable. If the variable has the Null Field property set to Yes, JAM automatically converts the null string to the one assigned by the widget properties.

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 Yes or the Justification property set to Right, the property is applied.

Date-time Column

If a column has a date data type, JAM formats the value before writing it to a JAM variable. If the variable has a date-time edit, JAM uses it. If the variable does not, JAM 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, 1994 10:05:03` would be formatted as `4/1/94 10:05`. When the message file default is used, JAM assumes a 12-hour clock.

For information on date-time formats, refer to the *Editors Guide* and the *Configuration Guide*.

Numeric Column

If a column has an integral type, JAM converts the value to a long. JAM then converts the value to ASCII and writes it to the variable, truncating any data longer than the destination variable.

If a column has a real type, JAM converts the value to a double. Before writing the value to a JAM variable, JAM examines the widget's Data Formatting and C Type properties to help determine the precision.

- Data Formatting⇒Numeric and C Type⇒Default

  If the value is less precise than the edit's 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 currency edit's maximum number of decimal places. Note that the round up, round down, or adjust option of the currency edit is applied.

- Data Formatting⇒None and C Type⇒Float/Double/Int/Long Int/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 currency edit's minimum number of decimal places, the value is padded to the minimum number of decimal places.
  - If the value is more precise than the minimum number of places, JAM compares the currency's maximum number of places and the C type's precision, and uses the less precise of the two.
  - If it uses the currency's maximum number of places, then it also uses the currency's round up, round down, or adjust option as well as any fill characters.
  - If it uses the C type precision, it adjusts by standard rounding to the precision.

- Data Formatting⇒None and C Type⇒Default

  The precision is taken from the data type being returned.

Refer to the *Editors Guide* for more information on currency formats.

Binary Columns

If a column has an binary data type, JAM sets the column type to be `DT_BINARY`. Before writing data to the widget, JAM checks the C Type property. If the C Type is Hex Dec, then JAM converts the binary data to a hexadecimal string. Otherwise, JAM 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 Hex Dec. Otherwise, incorrect binding might result.

Fetching Unique Column Values

By default, when a column is selected, JAM returns all values. There is also a command for displaying only a column's unique values,

```
DBMS [WITH CURSOR cursor-name ] UNIQUE column \
    [, column ... ]
```

JAM 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 12 illustrates the data in the `project` table.

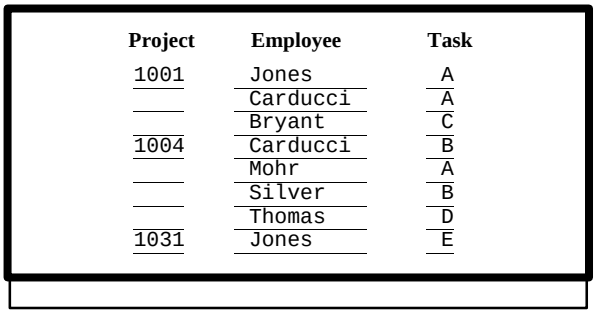
project_id	staff	task_code
1001	Jones	A
1001	Carducci	A
1001	Bryant	C
1004	Carducci	B
1004	Mohr	A
1004	Silver	B
1004	Thomas	D
1031	Jones	E

Figure 12. 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 13 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 13. The JAM layout is easier to read than the table layout.

Refer to Chapter 11 in the *Database 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 JAM variable or a text file.

```
DBMS [WITH CURSOR cursor-name] CATQUERY TO jam-variable \
   [SEPARATOR text] [HEADING [ON | OFF]] ]
```

```
DBMS [WITH CURSOR cursor-name] CATQUERY TO FILE filename \
   [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 both of these commands, refer to Chapter 11 in the *Database Guide*.

Writing Information to the Database

The following sections discuss how JAM passes data from an application to a database. The topics are the following:

- Colon preprocessing: using the colon preprocessor to put JAM values into SQL statements. Its forms are :*variable*, :+*variable*, and :=*variable*.
- Parameters: binding values to SQL parameters when executing a named cursor. Their form is ::*variable*.

Colon Preprocessing

JAM supports colon preprocessing as part of its standard JPL syntax. This standard colon preprocessing is described in the JPL section of the *Language Reference*. One or more colon variables can appear anywhere in a `DBMS` statement. One exception applies: 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, JAM'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. They are 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. Note that 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 the JPL statement, the following steps are performed:

1. The standard colon preprocessor replaces the variable `:+`*variable* with the value of *variable*.

2. The colon-plus processor examines the following widget properties to determine the variable's JAM type in order to know how to format the data:

   - C Type under Identity
   - Null Field under Format/Display
   - Data Formatting under Format/Display
   - Keystroke Filter under Input

3. If the JAM widget has a Data Formatting value set or if it is a character string, the processor formats the value according to engine-specific rules. Character strings are usually enclosed in quotation characters. For other format types, the processor calls a function to strip amount editing characters, such as dollar signs, from the value. Finally, the new value is returned to the JPL statement.

The following sections describe these steps in detail.

Step 1: Perform Standard Colon Preprocessing

JAM will search 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.

Note that when JAM is executing a screen entry function, JAM by default will search for *variable* in the LDB before searching the current screen. For more information on variables and their scope, refer to the JPL section of the *Language Reference*.

Step 2: Determine the Variable's JAM Type

A widget or LDB variable has exactly one JAM type. Since a variable may have more than one of the qualifying properties, JAM uses some precedence rules when assigning the JAM type by performing these checks:

1. Checks the Null Field property under Format/Display to see if the variable contains a null value.
2. Checks the C Type property setting under Identity. This property has the highest priority in determining the variable's JAM type.
3. Checks the Data Formatting property under Format/Display for Numeric settings indicating currency formatting and for Date/Time settings.
4. Checks the Keystroke Filter property under Input.

If the variable does not fall into one of these categories, it is assigned the JAM type of `FT_CHAR`.

The JAM types are:

`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`	

This assignment is explained in more detail in the following sections.

Null Field Property

First, JAM looks to see if the variable is a null value. If the value is null, it is not necessary to determine the JAM type. To determine if the value is null, JAM checks to see if the widget or LDB entry has the Null Field property set to Yes. If the value of the variable equals this null edit string, the processor replaces the value with the engine's null string. On most engines, it is the string `NULL`. For example, the widget named `rating_code` has the following properties set under Format/Display:

Property Name	Property Entry
Null Field	Yes
Null Text	*
Repeating	Yes

If you do not enter text in the `rating_code` widget, it is null and JAM displays the string, `****` as the widget contents. JAM's database driver would convert the string `****` to `NULL` (i.e., the value of the engine's null string) before passing it to the database engine.

If you enter text in the `rating_code` widget, the processor proceeds to determine the variable's JAM type from other widget properties described in the following sections.

blank numeric fields

It should be noted that if a numeric field is blank or empty, JAM substitutes `NULL` as the column's value for that field, even if the Null Field property is set to No. If the column has been specified as `NOT NULL` in the database, the engine will return an error.

C Type Property

Next, JAM looks at the C Type property under Identity. If the C Type property is not set to Default or Omit, it is used to assign a JAM type.

When you create a widget on the screen, JAM automatically assigns the Default C type to the widget. However, if you copy the widget from a repository imported from your database tables, the database importer assigns a C type based on the column's data type. You may also explicitly set a C type.

The JAM type for the C Type property values are as follows:

C Type	JAM Type
Char String	FT_CHAR
Hex Dec	FT_HEX
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 Property

Next, JAM examines the Data Formatting property under Format/Display. If set to Date/Time, the JAM type is DT_DATETIME. If set to Numeric, the JAM type is DT_CURRENCY.

Keystroke Filter Property

If the Data Formatting property is set None, JAM examines the Keystroke Filter property under Input:

- Digits Only values are assigned the JAM type FT_UNSIGNED.
- Yes–No values are assigned the JAM type DT_YESNO.
- Numeric values are assigned the JAM type FT_DOUBLE.

FT_CHAR Assignment

For all other widget and LDB variables, and for all JPL variables, JAM assigns FT_CHAR as the JAM type.

Type Conflicts

Beware of C type property setting that may conflict with other properties. For example, if a widget has a C Type setting of Int and a Data Formatting setting of Date/Time, its JAM type would be FT_INT. The Date/Time format would be

enforced for user entry but JAM's database drivers would not convert the date-time string into a format the engine would recognize.

Note: *You may also use the* `sm_ftype` *library function to determine a variable's JAM type. The assignments are the same as those in the table above, except for JPL variables. The library function* `sm_ftype` *returns 0, not* `FT_CHAR`*, for JPL variables.*

Step 3: Format a Non-null Value

Once a non-null variable's JAM type is determined, this classification is used to perform any necessary formatting before returning the formatted text to JPL.

DT_DATETIME Variables

If the JAM type is `DT_DATETIME`, the processor calls the support routine to format the text in the engine's default syntax for dates. Some support routines store a JAM date-time format string in the style of the engine. When formatting a field value, it may simply pass the format string and value to JAM's date-time routines to reformat the string. Other support routine may call a conversion function from the DBMS library to perform the task.

Of course, the actual result is dependent on the engine. For example, if the value in a date-time field is December 31, 1999 3:05 PM and the current engine is using the ORACLE support routine, JAM formats the date like this:

```
TO_DATE('31121999 150500', 'ddmmyyyy hh24miss')
```

If the engine is using the SYBASE support routine, however, JAM formats the date as follows:

```
'Dec 31,1999 3:5:0:000PM'
```

Some engines support more than one data type for date-time columns. For more information, refer to the Engine Notes section of the *Database Guide*.

FT_CHAR Variables

If the JAM type is `FT_CHAR`, 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, the number of embedded `quote_char`'s 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, JAM would format the field value

```
Ms. Penelope O'Brien
```

to

```
'Ms. Penelope O''Brien'
```

JAM would format the field 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 Engine Notes section of the *Database Guide*.

FT_HEX Variables

If the JAM type is `FT_HEX`, JAM 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 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 change the Widget Type to Multitext and set the Word Wrap property to Yes.

FT_Numeric and DT_CURRENCY Variables

For the remaining JAM types, the processor calls the JAM function `sm_strip_amt_ptr` to strip editing characters from the numerical string. The function strips all non-digit characters except for an optional leading negative sign and a decimal point. Refer to the *Language Reference* for more information on `sm_strip_amt_ptr`. The colon preprocessor does not use precision edits when formatting numeric values.

For example, JAM would format

```
$500,000.00
```

as

```
500000.00
```

JAM would format

```
(-89.003)
```

as

```
-89.003
```

It would format

```
001-02-0003
```

as

```
001020003
```

If you wish to preserve embedded punctuation in numeric fields, set the widget's C Type property to Char String.

For more information, refer to the Database Drivers section of the *Database Guide*.

Empty Numeric Variables

If a numeric or currency field is empty or blank, JAM substitutes `NULL` as the column's value for that field, even if the Null Field property is set to No. If the column has been specified as `NOT NULL` in the database, the engine will return 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 permit you to select rows where a column value is either known or unknown (i.e., NULL), use the colon-equal processor. For example,

```
DBMS SQL SELECT * FROM titles WHERE rating_code :=rating_code
```

If the widget named `rating_code` has the following properties set under Format/Display:

Property Name	Property Entry
Null Field	Yes
Null Text	*
Repeating	Yes

JAM would format the value

```
PG
```

as

```
= 'PG'
```

thus executing

```
SELECT * FROM titles WHERE rating_code = 'PG'
```

It would format the field's "null" value

```
****
```

as

```
IS NULL
```

thus executing

```
SELECT * FROM titles WHERE rating_code IS NULL
```

Examples

Widget with Default Settings

The current screen has a widget named `last_name`. It has the following property settings:

Property Category and Name	Property Entry
Identity⇒C Type	Default
Input⇒Keystroke Filter	Unfiltered
Format/Display⇒Data Formatting	None
Format/Display⇒Null Field	No

With these settings, the JAM type is `FT_CHAR`. If the widget `last_name` contained the text `D'Angelo` when the following statement was executed:

```
DBMS SQL SELECT * FROM customers
   WHERE last_name = :+last_name
```

JAM would pass the query

```
SELECT * FROM customers WHERE last_name = 'D''Angelo'
```

If the widget `last_name` was empty, JAM would pass the empty string, not the null string as follows:

```
SELECT * FROM employee WHERE last_name = ''
```

Null conversion is performed only on variables with a null field edit.

Widget with Date/Time and Null Field Settings

If the current screen contains a widget `member_date` with the following property settings:

Property Category and Name	Property Entry
Identity⇒C Type	Default
Input⇒Keystroke Filter	Digits Only
Format/Display⇒Data Formatting	Date/Time
Format/Display⇒Format Type	Various
Format/Display⇒Null Field	Yes
Format/Display⇒Null Text	00/00/00

Assume that back slash characters are an edit mask applied to the field. Since a Date/Time setting has a higher precedence than the Keystroke Filter setting, the JAM type for this widget is `DT_DATETIME`. If you enter the date `12/31/93` and execute the following function,

```
DBMS SQL WITH CONNECTION oracle_conn \
   INSERT INTO customers (cust_id, last_name member_date) \
      VALUES (:+cust_id, :+last_name, :+member_date)
```

and the engine, for example, were ORACLE, JAM would pass the following statement to the engine:

```
INSERT INTO customers (cust_id, member_date) VALUES \
       (43, 'D''Angelo', \
        TO_DATE('31121991 000000', 'ddmmyyyy hh24miss'))
```

If you did not enter any text in the widget `member_date`, its contents would be `00/00/00` and JAM would pass the following statement to the engine:

```
INSERT INTO customers (cust_id, last_name, member_date) \
   VALUES (43, 'D''Angelo', NULL)
```

Widget with Digits Only and Char String Settings

Often it is useful to use the Digits Only setting in the Keystroke Filter property for widgets that accept numeric values, such as a telephone number. If this is the only edit on the field, the colon-plus processor will format the widget's value as an unsigned integer, removing embedded punctuation and leading zeros. However, if you reset the C Type property to Char String, the colon-plus processor will format the field's contents as a character string, preserving embedded punctuation and leading zeros.

The property settings are as follows:

Property Category and Name	Property Entry
Identity⇒C Type	Char String
Input⇒Keystroke Filter	Digits Only
Format/Display⇒Data Formatting	None
Format/Display⇒Null Field	No

For example, if you enter `00912` in the `postal_code` widget and execute the following statement:

```
DBMS SQL SELECT * FROM customers \
   WHERE postal_code = :+postal_code
```

JAM would pass the following query to the engine:

```
SELECT * FROM customers WHERE postal_code = '00912'
```

Note that if the Keystroke Filter property is set to Digits Only but the C Type is not set to Char String, the following query would be passed to the engine:

```
SELECT * FROM customers WHERE postal_code = 912
```

Widget with C Type of Hex Dec

Setting the widget's C Type property to Hex Dec is one method used to fetch binary values to JAM screens. With this setting, when binary data is fetched in a SQL `SELECT` statement, JAM converts the binary value to a hexadecimal string. If any subsequent database updates use this data, it is converted back to a binary format before being passed to the database engine.

Property Category and Name	Property Entry
Identity⇒C Type	Hex Dec

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 (e.g., Progress and SYBASE), JAM internally supports cursors with parameters.

When JAM executes a `DECLARE CURSOR` statement, it scans the statement for parameters. For all engines, JAM recognizes the following syntax to be a parameter:

`::`*parameter*

Note that 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. JAM also recognizes these engine-specific forms.

If JAM finds a parameter, it sets up a data structure for it. It will attempt to find a value for the parameter when the cursor is executed. Parameters may 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 ...]*

JAM 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, JAM assumes it is a variable and performs variable substitution and formatting. Values and parameters may 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 may also 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
```

Note that `p3`, `p1`, and `p2` are not JAM variables but `role`, `title_id`, and `actor_id` are. JAM uses the values of `role`, `title_id`, and `actor_id` to execute the `INSERT`. To supply a literal value to the `INSERT`, put the value in quotes:

```
DBMS WITH CURSOR b_cursor EXECUTE \
   USING p3=role, p1="89", p2=actor_id
```

JAM 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 often improves the efficiency of the application, especially when a query is executed several times. On engines where JAM simulates support, such as SYBASE, the use of parameters will be less efficient. However, the convenience and the greater ease of portability may compensate for the additional processing.

Parameter Substitution and Formatting

An *arg* in a `USING` clause can be:

- A quoted string
- A JAM variable

Colon-plus processing is not necessary because JAM automatically formats the value of parameter variables. If the variable is an array, an occurrence number may

be given. If no occurrence is given, JAM concatenates all the non-empty occurrences in the array, separating the occurrences with a single space. Substrings are not permitted.

For each cursor, JAM maintains binding information. When a cursor's statement uses parameters, JAM stores the names of the parameters. When a cursor is executed, JAM compares the values in the `DBMS EXECUTE` statement with the binding information from the cursor's declaration. This permits both positional and explicit binding.

JAM uses a data structure to store the formatted text and JAM type of *arg*. If *arg* is not quoted, JAM assumes it is a variable and calls `sm_ftype` to determine the variable's `ftype` code and flags. Like the colon-plus processor, the binding routine distinguishes between empty and null variables; a variable is null if it the Null Field property is set to Yes and the variable contains the null string.

If the `ftype` is `DT_DATETIME`, JAM calls the support routine to convert the value to a binary date-time value. Refer to the discussion of `DT_DATETIME` (on page 248) for more information.

No processing is done on the values of `FT_CHAR` variables or quoted strings.

For all other types, JAM strips characters other than digits, the decimal point, and a leading negative sign from the value.

Below are some examples showing 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]
```

Examples

Using Currency Formats

If the current screen contained a widget named `rent_amount` with the following property settings:

Property Category and Name	Property Entry
Identity⇒C Type	Default
Input⇒Keystroke Filter	Numeric
Format/Display⇒Data Formatting	Numeric
Format/Display⇒Format Type	Local Currency

For this widget, the JAM type would be `DT_CURRENCY`.

If you entered the total `$1,000.99` in this widget and executed the following statements:

```
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 would execute

```
SELECT * FROM customers WHERE rent_amount > 1000.99
```

Using Multiline Text Widgets and Arrays

If the current screen contained a widget named `notes` which is a null field and an array, its property settings would be:

Property Category and Name	Property Entry
Identity⇒Widget Type	Text
Identity⇒C Type	Default
Geometry⇒Array Size	3
Input⇒Keystroke Filter	Unfiltered
Format/Display⇒Data Formatting	None
Format/Display⇒Null Field	Yes
Format/Display⇒Null Text	

For this widget, the JAM type would be `FT_CHAR`.

If you executed 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 array is empty, the DBMS would execute

```
INSERT INTO customers (cust_id, notes) VALUES (123, '')
```

If, however, the array contained text, JAM would concatenate the non-empty occurrences into one long string which the DBMS would insert 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.')
```

The same behavior would apply if the Widget Type is Multitext and the Word Wrap property is set to Yes.

Error Processing in Database Applications

JAM provides global variables and hook functions to help you manage errors in a database application. A default error handler is installed to display messages from JAM's database drivers and from the database engine. In addition, you can write and install a customized error handler for your application either in JPL or in C. This need not be complicated. With a single JPL procedure, you can change the way errors are handled and control which messages are displayed.

This chapter discusses:

- The behavior of the default error handler.
- The global variables containing error and status information from JAM's database drivers and from the database engine.
- The hook functions available for writing your own error handlers.
- The installation of an error handler for your application.

Default Error Handler

JAM installs a default error handler which is used when executing `DBMS` commands. If an error occurs, the default error handler displays the following information in a dialog box:

- An engine-independent error message from JAM'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.

Figure 14. Sample error message from the default error handler.

For all errors, the default error handler returns a –1 to its caller. If an error occurs while executing JPL, JAM aborts the JPL procedure where the error occurred. An aborted JPL procedure always returns –1 to its caller.

An application may override the default handler by installing its own function to handle errors. It may also install an exit function to process all error and status information and to display this information in the application. This is covered in the following sections.

Using Variables with Error and Status Information

JAM supplies several pre-defined variables where it stores error and status data for the application. After executing a `DBMS` statement, JAM updates these variables with any error, warning or status information returned by the engine. In addition to the engine-specific codes and messages, JAM also supplies engine-independent codes and messages to the variables `@dmretcode` and `@dmretmsg`. These messages are defined in `dmerror.h`.

The global variables available through JAM's database interface are automatically defined at initialization. All the global variable names used in the database interface begin with the characters `@dm`. Since the character `@` is not permitted in user-defined JAM variables, these variables will never conflict with any screen, LDB or JPL variables defined by your application.

These variables and their values are available to JPL commands and to JAM library functions like the _n_ variants of `sm_getfield` and `sm_fptr`.

The variables are automatically maintained by JAM. Before executing a `DBMS` statement, JAM clears the contents of all its global variables. After executing the statement and before returning control to the application, JAM updates the variables to indicate the current status.

Variable	Description
`@dmretcode`	The status of the last executed `DBMS` statement. Its value is 0 or one of the codes defined in `dmerror.h`.
`@dmretmsg`	A message describing the status of the last executed `DBMS` statement. Its value is either empty or one of the messages from the JAM message file. If `@dmretcode` is 0, this variable is empty.
`@dmengerrcode`	An engine-specific error code for the last executed `DBMS` statement. Its value is 0 or an engine-specific code. If 0, the engine did not detect any errors.
`@dmengerrmsg`	An engine-specific error message for the last executed `DBMS` statement. If `@dmengerrcode` is empty, this variable is also empty.
`@dmengwarncode`	An engine-specific warning code or bit setting for the last executed `DBMS` statement. If empty, the engine did not detect any warning conditions.
`@dmengwarnmsg`	An engine-specific warning message describing the warning code for the last executed `DBMS` statement. If `@dmengwarncode` is a byte or is blank, this variable is also empty.
`@dmengreturn`	The return code from the last executed stored procedure. Its value is either blank or an integer. If blank, the engine did not supply a return code.
`@dmrowcount`	The number of rows fetched to JAM variables by the last `SELECT` or `CONTINUE` statement. On some engines, it can also contain the number of rows affected by an `INSERT`, `UPDATE` or `DELETE` statement.
`@dmserial`	An engine-generated value for a serial column. Its value is 0 or an appropriate serial value for the column.

For more information on these variables, refer to the Database Reference and the Database Drivers sections of the *Database Guide*.

Using the Error Hook Functions

JAM provides the following hooks for database error functions:

ONENTRY	This function is called before executing any DBMS command from JPL or C.
ONEXIT	This function is called after executing any DBMS command from JPL or C.
ONERROR	This function is called if an error occurs while executing any DBMS command from JPL or C.

The hook functions may be written in JPL or C.

A JPL hook function is installed as follows:

```
DBMS { ONENTRY | ONEXIT | ONERROR } JPL entry-point
```

where *entry-point* is an entry point to a JPL module. An entry point may be a procedure name or a file name. Refer to the JPL section of the *Language Reference* for more information.

A C hook function is installed as follows:

```
DBMS { ONENTRY | ONEXIT | ONERROR } CALL function
```

where *function* is a prototyped function. A prototyped function appears on JAM's PROTO_FUNC list. It must be prototyped with three arguments: two strings and an integer. For example:

```
static struct fnc_data pfuncs[] =
{
       {sm_flush()”,flush, 0, 0, 0, 0},
       ...
       {function(s,s,i)”, function, 0, 0, 0, 0  },
}
```

For more information on prototyped functions, refer to Chapter 8.

To turn off an error hook function, execute the command with no arguments. For example:

```
DBMS ONERROR
```

For more information and examples of each hook function, refer to Chapter 11 in the *Database Reference*.

ONENTRY Function

Before executing a DBMS command from JPL or C, JAM executes the application's installed ONENTRY function. An ONENTRY function is useful for logging or debugging statements. You may also use an ONENTRY function to modify the JAM environment, for instance, to remap cursor control keys or change protection edits on widgets.

ONEXIT Function

After executing a DBMS command from JPL or C, JAM executes the application's installed ONEXIT function. An ONEXIT function is useful for logging or debugging statements. Like ONENTRY, you may use an ONEXIT function to modify the JAM environment, for instance, to remap cursor control keys or change protection edits on widgets. This function is also useful for checking error and status codes after each command.

ONERROR Function

If an error occurs in the database driver while executing a DBMS command from JPL or C, JAM executes the application's installed ONERROR function. An ONERROR function can display the values of the global error variables. It may also display the text of the command that failed. The application may also use this function to log error information in a text file.

An ONERROR function overrides JAM's default 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 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, JAM calls the ONEXIT function, then the ONERROR function.

Function Arguments

The error hook functions receive three arguments:

1. A copy of the first 255 characters of the command line. If the command was executed from JPL, this is the first 255 characters after the JPL command word DBMS or DBMS SQL.

2. 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.

3. A context flag identifying why this function was called. For an `ONENTRY` function, its value is 0. For an `ONEXIT` function, its value is 1. For an `ONERROR` function, its value is 2.

Return Codes

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. To ensure compatibility with future releases, 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, JAM will abort the JPL procedure where the error occurred. 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 JAM's database interface while executing JPL, a non-zero return code aborts the JPL procedure where the error occurred. The procedure's caller (either JAM or another JPL procedure) gains control. If the return code is 0, the JPL procedure resumes control; JAM will execute the next statement in the JPL procedure.

If an error occurs in JAM'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.

Installing an Error Handler

It is recommended that you install an error handler for your application. The error handler may be written in either in JPL or C. This allows you to customize the error messages appearing in your application. You can use any of the global variables as part of this error handler. For example, it may use `@dmretmsg` to display a message from JAM's database driver or `@dmengerrmsg` to display an engine-specific error message. It may 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 JAM's database drivers:

- Syntax or logic error in a `DBMS` statement.

  Some examples are 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 SQL` statement. These errors are detected by JAM's database driver. These errors update the global variables `@dmretcode` and `@dmretmsg`.

- Engine error.

  Some examples are 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 JAM's database driver. These errors update the global variables `@dmretcode`, `@dmretmsg`, `@dmengerrcode`, `@dmengerrmsg`.

Note that there are also JAM and JPL errors which are not a part of JAM's database driver. A JPL procedure may fail because of JPL syntax or colon preprocessing errors. If a JPL error occurs, JAM displays 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 JAM errors are detected and corrected during application development. The only time that an application may need special handling for these errors is during transaction processing. This is discussed in Chapter 17.

An `ONERROR` function overrides JAM's default 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.

Developers using JPL are encouraged to use 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, JAM calls the ONEXIT function, then the ONERROR function.

Example

This procedure first checks if the error is DM_ALREADY_ON. In this case, it simply displays a message and returns 0. For all other errors, it checks for an engine error code. If there is an engine error, it displays the statement and engine-specific error message. For any other errors, it displays the standard JAM message.

```
proc screen_entry
    DBMS ONERROR jpl dbi_error_handler
    .
    .
    .
return

proc dbi_error_handler (stmt, engine)

    if (@dmretcode == DM_ALREADY_ON)
    {
        msg emsg ”You are already logged on.”
        return 0
    }

    if (@dmengerrcode != 0)
    {
        msg emsg @dmretmsg ”%N” ”Statement :stmt” ”%N” \
            ”:engine Error :@dmengerrcode :@dmengerrmsg”
    }
    else
    {
        msg emsg ”Application Error:  :@dmretmsg ” \
            ”See the DBA for assistance.”
    }

return 1
```

Database Transactions

This chapter describes database transactions and JAM's support of a database engine's transaction processing.

Introduction to Transactions

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` application, these are two of the 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 involves an update to the `customers` table for the rental amount and the number of rentals. The third statement updates the `tapes` table in order to increase by 1 the number of times the tape has been rented and to change the status of the rental.

- A new video transaction.

  This transaction involves inserts 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.

Transaction processing is sometimes a difficult topic for new developers. For one thing, transaction processing is very engine dependent and thus it requires a clear understanding of the engine's behavior. For another, transaction processing in a JAM application requires careful error processing. For some errors, the application must explicitly tell the engine to undo the transaction. The application must test for these errors.

Engine-specific Behavior

As noted earlier, transaction processing is not implemented consistently among SQL databases. Developers should review the documentation on transaction processing supplied by the database vendor before using JAM features.

Generally, transaction processing falls into two types: those that support explicit transactions and those that support auto transactions. An explicit transaction starts with a `BEGIN` statement; an auto transaction generally starts with the first recoverable statement after a logon, `COMMIT`, or `ROLLBACK`. Usually an engine supports either explicit transactions or auto transactions, but not both.

On engines supporting explicit transactions, each `COMMIT` or `ROLLBACK` must have a matching `BEGIN`. On engines supporting autocommit modes, the application may 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 engines automatically commit or rollback the transaction. Others may leave the database in an inconsistent state. Under no circumstances should the application use the engine's default behavior to terminate a transaction.

The use of explicit rollbacks and commits:

- Protects the integrity of the database.
- Makes new and updated data available to the rest of the application and other users at the logical end of the transaction.
- Releases locks set on tables by the transaction which would otherwise be held until the connection closes, permitting the rest of the application or other users to begin new transactions on the tables.

- Reduces the chances for unrelated operations to interfere with one another.
- Produces applications which are less database-dependent.

Finally, although vendors supply commands for transaction processing in their SQL language, you should use `DBMS COMMIT`, `DBMS ROLLBACK`, and other transaction commands provided by JAM. Using `DBMS SQL` to specify engine-specific commit and rollback processing is *not* recommended. Using the `DBMS` versions permits JAM 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 JAM 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 JAM before a statement is passed to the engine. An error detected by JAM rather than the engine is usually the result of a development or maintenance error rather than bad user input (e.g., a statement's colon-plus or binding variable cannot be found because a JAM field was renamed). While these errors should be rare, the application should provide handling for them.

If the transaction processing is done with the C library functions provided by JAM's database drivers, error codes from JAM are returned to the calling function, either directly or via an installed error handler. If a transaction requires very sophisticated error handling, it may be easier to use these JAM 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 JAM 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, JAM 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 SQL 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 section in the *Database Guide* for a list and description of the supported transaction commands for each engine.

SECTION FOUR

SQL Generation

SQL Generator

SQL (Structured Query Language) is the database manipulation language used by many relational database management systems.

Earlier chapters in this manual provide information about including SQL statements as part of your JPL procedures or C functions. This chapter discusses how the SQL generator builds SQL statements from various screen and widget properties.

In the current release, the major interface to the SQL generator is the transaction manager. The standard transaction models used by the transaction manager call the SQL generator to create SQL statements at runtime. This chapter briefly discusses the transaction manager commands corresponding to the major SQL statements.

For basic information about SQL and about constructing SQL statements, refer to Chapters 2 and 3 in the *Database Guide*. For information about including SQL statements in JPL procedures, refer to Chapters 14 and 15 in this manual.

SQL Generation Overview

This chapter contains a major section for each type of SQL data manipulation statement: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. The syntax of the statement is presented first, followed by a discussion of each element in the statement. Since some of the guidelines used for database tables and columns are the same for all SQL statements, those guidelines are listed here.

Specifying Tables

The table view widgets in JAM contain most of the database table information. The following guidelines apply to database tables:

- The table name used in the SQL statement is the Table property, found in the Database category of the table view properties.
- In order for a database table to be included in a SQL statement, a table view corresponding to that database table must exist on the screen.
- If the screen has more than one table view, links defining the relationship between table views must also exist on the screen for automatic SQL generation to occur.
- The table view must be Updatable in order to generate SQL `INSERT`, `UPDATE` and `DELETE` statements for that table.
- Since you can apply a transaction manager command to one or more table views, all the table views on a screen do not necessarily participate in each command.

Specifying Columns

The widgets in each table view generally correspond to the database columns. The following guidelines apply to database columns:

- The column name is the Column property, found in the Database category of the widget properties.
- A correlation name (generally *table-name.column-name*) is used in the SQL statement unless the Expression properties under the Database heading 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 table, a widget corresponding to that column must exist in the table view associated with that table.
- In order to participate in SQL `SELECT`, `INSERT` and `UPDATE` statements, the corresponding widget properties, Use In Select, Use In Insert, and Use In Update, must be set to Yes. This is the default setting.

Generating SQL in the Transaction Manager

The transaction manager is the major interface to the SQL generator in the current release. When you use the transaction manager, the SQL statements are automati-

cally generated from the various property settings. You can change the SQL statements by editing the properties or by adding hook functions to handle certain transaction manager requests. For more information on writing hook functions, refer to Chapter 22.

The following table outlines which transaction manager commands are needed to generate the different types of SQL statements:

SQL Statement	Transaction Manager Command
`DELETE`	Generated via `SAVE` command after rows have been deleted or cleared of data in update mode. Table view must be updatable.
`INSERT`	Generated via `SAVE` command after new rows have been 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 has been modified in update mode. Table view must be updatable.

Example Tables

In order to illustrate the SQL generation, the examples in this chapter use the following database tables which are part of a database called `vacation`:

```
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)
)
```

Please note that the SQL examples in this chapter may not match the SQL generated by JAM. This can occur for the following reasons:

- The SQL generated by JAM is for a specific database engine.
- The order of items in the statements sometimes depends on the order in which widgets are added to the screen. In this case, the order should not affect the results.

The examples contain tables listing which properties you need to set for each widget and table view in order to produce the necessary SQL. Generally, the properties are located in the Database category of the Properties window.

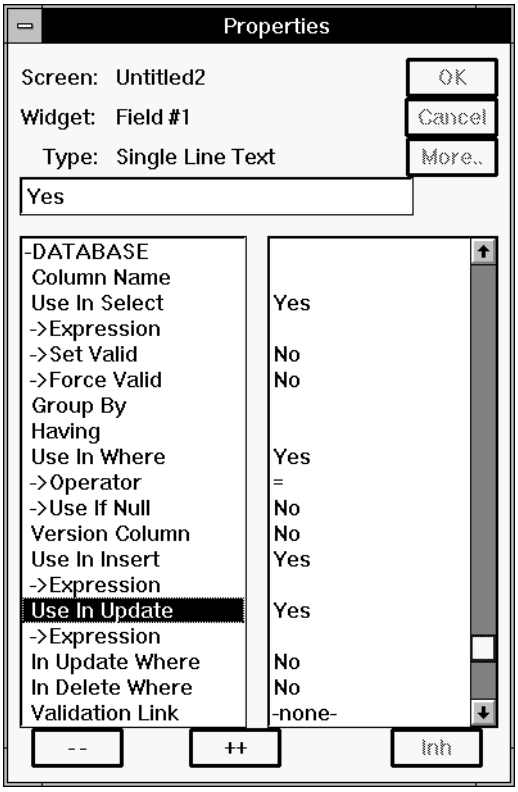


Figure 15. The Database category in the Properties window.

SELECT Statement Overview

The SQL generator generates one SQL `SELECT` statement per server view. Recall that 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.

Here is the syntax of the SQL `SELECT` statements that can be generated by JAM's 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 ]
```

The following table lists the major elements in a SQL `SELECT` statement, and briefly describes how to set properties to trigger generation of those elements. More detailed information is presented in the sections following the table.

SQL Element	Property Settings
distinct-keyword	For the table view, set the Database⇒Distinct property to Yes.
select-list	For each widget in the server view, the value in the Database⇒ColumnName property unless an expression has been set under the Use In Select property. The Use In Select property must also be set to Yes.
table-name	For the table view, the value in the Database⇒Table property.
`WHERE` clause	In the applicable widgets, set the Use In Where property to Yes and choose the desired Operator property. The following operators are available: `=, <>, <, <=, >, >=, in, like, like%, %like%, not in, not like, not like%, not %like%` Joins: For the link, set the Transaction⇒Type property to Server. Also, the 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 the applicable widgets, enter the column name in the Database⇒Group By property.
`HAVING` clause	For the applicable widgets, enter the search condition in the Database⇒Having property.
`ORDER BY` clause	For the table view, enter the widget name(s) associated with the column name(s) in the Database⇒Sort Widgets property, followed by `ASC` or `DESC`.

Other SQL elements that can be part of a SQL `SELECT` statement include:

SQL Element	Property Settings
Aggregate functions	For the applicable widgets, enter the aggregate function in the Expression section of the Database⇒Use In Select property.

SQL Element	Property Settings
`BETWEEN` predicate	Use hook function to call the function `dm_gen_change_select_where`.
`EXISTS` clause	Use hook function to call the function `dm_gen_change_select_where`.
`IN` clause	For the applicable array, set the Use In Where property to Yes and select the `In` operator. On the screen, enter values in the array before executing the `SELECT` statement.
`LIKE` predicate	For the applicable widgets, set the Use In Where property to Yes and select the one of the Like operators. On the screen, enter a value in the specified widget before executing the `SELECT` statement.
Null values	For the applicable widgets, set the Database⇒Use If Null property to Yes. Under Format/Display, set the Null Field property to Yes and enter the Null Text to be displayed on the screen.
Operators	Can only be set for `WHERE` clauses in `SELECT` statements. For the applicable widgets, set the Use In Where property to Yes and choose the desired Operator.
Stored procedures	Use hook function.
Subqueries	Use hook function to call the function `dm_gen_change_select_where`.

If a desired SQL statement cannot be generated automatically, you can write a transaction hook function either to supply the custom SQL or to call the SQL modification functions. For more information on writing transaction hook functions, refer to Chapter 22.

Setting the DISTINCT keyword

If a table view's Distinct property is Yes, then JAM supplies the correct word for the database, either `DISTINCT` or `UNIQUE`, and applies it to the server view.

Setting the Select List

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 Use In Select⇒Expression property, if set, or the widget's Column Name.

If the widget's Column Name is used, it appears in the following format:

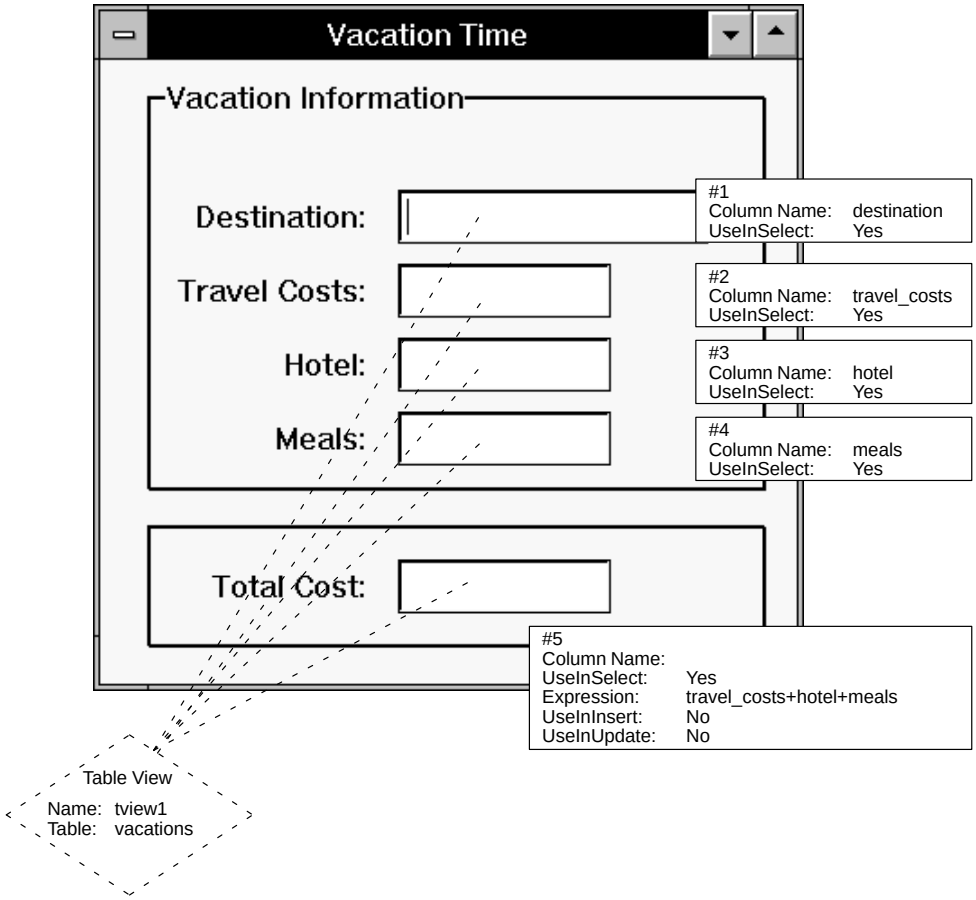
table-view-name.column-name

Example 1

Get the total cost of each vacation. The desired SQL is:

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
   FROM vacations
```

Create five widgets, and make them members of a table view associated with the table `vacations`. For Widget 5, set the Use In Insert and Use in Update properties to No which prevents this derived column from being included in `INSERT` and `UPDATE` statements.



Setting the Table List

The *table-list* is a comma-separated list of all of the Database⇒Table properties of the table views in the server view. For each table, a correlation name, or alias, pairs the database table with its associated table view name.

If a database table is imported to the repository by a user that is not the owner of the table, two table view properties, Identity⇒Name and Database⇒Table, also list the owner name. In this case, the owner name appears in the table list in the format:

owner.*table-name*

Setting the Where Condition

Using an Operator

The *where-condition* is derived from the widgets whose Use In Where property is Yes. If more than one widget has this setting, the `AND` keyword is used to join the

conditions. A *where-condition* compares data entered in the widget with data in the database column. The column name is derived from the widget's Column Name property. The comparison operator is the value of Use In Where⇒Operator. The supported operators are:

=	<	>	in	like	like%	%like%
<>	<=	>=	not in	not like	not like%	not %like%

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.

Example 2

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
```

Create the widgets described in Example 1. Add the following values to Widget 1 which is associated with the column `destination`:

Object and Property	Value
Widget 1	
Database⇒Use In Where	Yes
Database⇒Use In Where⇒Operator	=

Example 3

If you change the Operator to `like%`, you can use the pattern matching capability of the database to search for the desired destination:

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
   FROM vacations
   WHERE destination LIKE destination%
```

Object and Property	Value
Widget 1	
Database⇒Use In Where	Yes
Database⇒Use In Where⇒Operator	like%

On the screen, enter an expression, for example `Lon`, as the value for `destination` before choosing the `SELECT` or `VIEW` command. The destinations beginning with those letters will be displayed on the screen.

Example 4

If you change the Operator to `in` and make the widget an array, a series of destinations can be entered for database searches.

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
   FROM vacations
   WHERE destination IN (destination, destination, ...)
```

Object and Property	Value
Widget 1	
Geometry⇒Array Size	5 (any integer value is valid)
Database⇒Use In Where	Yes
Database⇒Use In Where⇒Operator	in

The array can be any size allowing you to enter a name in each occurrence before choosing the `SELECT` or `VIEW` command.

Selecting Null Values

Normally, widgets whose data is blank or null do not contribute to the *where-condition* (where null is defined as setting the widget's Null Field property to Yes). To force these widgets to contribute, set Use In Where⇒Use If Null to Yes. In that case, blank data will be treated as a null database values and a `WHERE` clause will be generated.

The text of the `WHERE` clause depends on the setting for the Operator property. If the operator is `=`, then the text is `WHERE` *column* `IS NULL`. If the operator is `<>`, then the text is `WHERE` *column* `IS NOT NULL`.

If the property Use In Where⇒Use If Null is set to No (the default), then no `WHERE` clause will be generated for the `SELECT` statement if `destination` is null or blank.

Example 5

Select rows where the values in a column are null. The desired SQL is:

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
   FROM vacations
   WHERE hotel IS NULL
```

For Example 1, add the following settings to Widget 3 which is associated with the column `hotel`.

Object and Property	Value
Widget 3	
Database⇒Use In Where	Yes
Database⇒Use In Where⇒Operator	=
Database⇒Use In Where⇒Use If Null	Yes

Using a Select Expression

If the Database⇒Use In Select⇒Expression contains an expression and the Database⇒Column Name is blank, the select expression cannot be used in the *where-condition* unless you use one of the SQL modification functions.

Setting the Group-by List

Automatic GROUP BY Clause

The SQL generator automatically builds a `GROUP BY` clause if any widget's Use In Select⇒Expression property uses one of the following 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 provided the widget's Use In Select property is set to Yes and the Use In Select⇒Expression is not considered an aggregate.

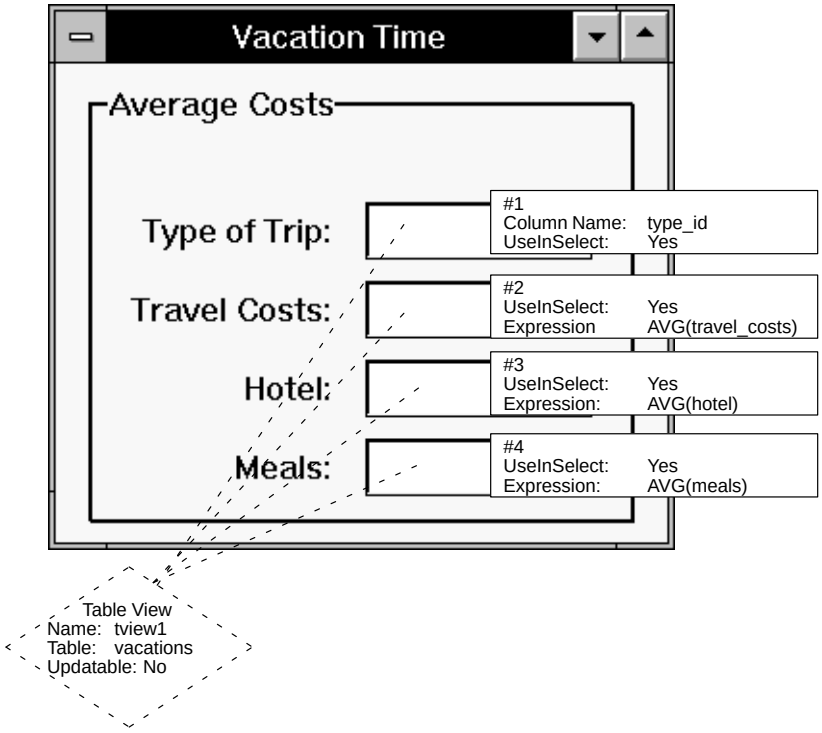
If one widget contains an automatically detected aggregate function, and another widget contains an undetected aggregate function, then JAM will erroneously add the other widget's column name to the *group-by-list*. Therefore, that widget's Database⇒Column Name should be blank.

Example 6

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
```

Create four widgets that are members of a table view associated with the table `vacations`. The table view property Transaction⇒Updatable should be set to No to prevent update and insert attempts. Note that the keywords `AVG`, `COUNT`, `SUM`, `MIN`, and `MAX` are not case-sensitive.



Specifying a GROUP BY Clause

When the SQL generator cannot detect the presence of an aggregate function in one of the widgets' Use In Select⇒Expression, a widget's Database⇒Group By property must be set. Enter the names of the columns whose values will be used to group the data.

Example 7

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
```

Create two widgets that are part of a table view associated with the table `vacations`. The table view property Transaction⇒Updatable should be set to No to prevent update and insert attempts. The Group By property must be set explicitly

because Widget 2 contains an aggregate function that is not automatically detected by the SQL generator.

Object and Property	Value
Widget 1	
Database⇒Column	`type_id`
Database⇒UseInSelect	Yes
Database⇒Group By	`type_id`
Widget 2	
Database⇒UseInSelect	Yes
Database⇒UseInSelect⇒Expression	`STDDEV(travel_costs+hotel+meals)`
Table View	
Database⇒Table	`vacations`
Transaction⇒Updatable	No

Specifying Multiple Columns

Another use of the Database⇒Group By property is to specify multiple column names, including columns not included in the *select-list*. The following example demonstrates this.

Example 8

Get the average net 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
```

Create two widgets that are a part of a table view associated with the table `vacations`. Even though JAM automatically detects the presence of an aggregate

function (AVG), the Group By property needs to be set because none of the widgets in the table view correspond to the type_id column.

Property	Value
Widget 1	
Database⇒Column	travel_costs
Database⇒UseInSelect	Yes
Widget 2	
Database⇒UseInSelect	Yes
Database⇒UseInSelect⇒Expression	AVG(travel_costs+hotel+meals)
Database⇒GroupBy	type_id
Table View	
Database⇒Table	vacations
Transaction⇒Updatable	No

Setting the Having Condition

The *having-condition* 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 whose Database⇒Having property is not empty. If more than one widget in the server view has this setting, the AND keyword is used to join these conditions.

Example 9

Get the average vacation cost, grouped by type. 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
```

Create the widgets described in Example 6. Complete the following changes to Widget 4.

Object and Property	Value
Widget 4	
Database⇒Having	`AVG(travel_costs+hotel+ meals) < 1000`

Setting the Order-by List

The *order-by-list* sorts the result rows according to the values in specified columns. The *order-by-list* is built from the Database⇒Sort Widgets property of the table view. To specify a sorting order, enter a list of *widget* names and an optional order specifier (case-insensitive). The valid order specifiers are:

- `DESC` Descending order
- `ASC` Ascending order

When the SQL is generated, JAM specifies the sorting order in a manner acceptable to the database engine. The order specifier should be separated from the widget name by white space. If no order specifier is entered following a widget name, then ascending order is assumed. If more than one widget is specified, each widget should be on a separate line. The SQL generator uses the widget name to determine the associated column or select expression to be sorted.

Example 10

Get the total cost of each vacation, ordered by cost in descending order. The desired SQL is:

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
   FROM vacations
   ORDER BY travel_costs+hotel+meals DESC
```

Make the following additions to the widgets described in Example 1. A name is entered for Widget 5 since the widget name, not the column name, is specified in the Sort Widgets property.

Object and Property	Value
Widget 4	
Identity⇒Name	`net_cost`
Table View (`vacations`)	
Database⇒Sort Widgets	`net_cost desc`

Generating SELECT Statements for Multiple Database Tables

The examples so far have dealt with only one database table at a time. This section illustrates how JAM can retrieve information from multiple database tables for the same application screen. The number of `SELECT` statements issued by the SQL generator depends on the type of link set in the link properties. For table views specified with server links, JAM issues a single statement, with a join in the `WHERE` clause. For table views specified with sequential links, JAM issues multiple statements using values fetched in the parent table view to create the where condition in the child table view.

Specifying Joins in the Where Condition

For equi-joins, joins where the operator is = (which includes self-joins), JAM automatically generates the necessary SQL. It does not generate SQL for other types of joins, such as outer joins. The information needed to build the join is found in the link's properties. The link between the joined table views specifies Server as the type of link and contains the names of the columns included in the `WHERE` clause.

You can specify the joining columns by selecting the link's Transaction⇒Relations property. The Relations dialog box enables you to enter or modify parent and child column names. You must also specify the relationship between the parent and child columns as `join`. For each join relation specified, the where condition will include one expression of the form

```
parent_table.parent_column = child_table.child_column
```

Note that the link's Transaction⇒Relations property also accepts field names, but these are valid only for sequential links (see below).

If there are multiple joined columns, then the expressions will be chained with the keyword `AND`. If more than one table view in the server view represents the same

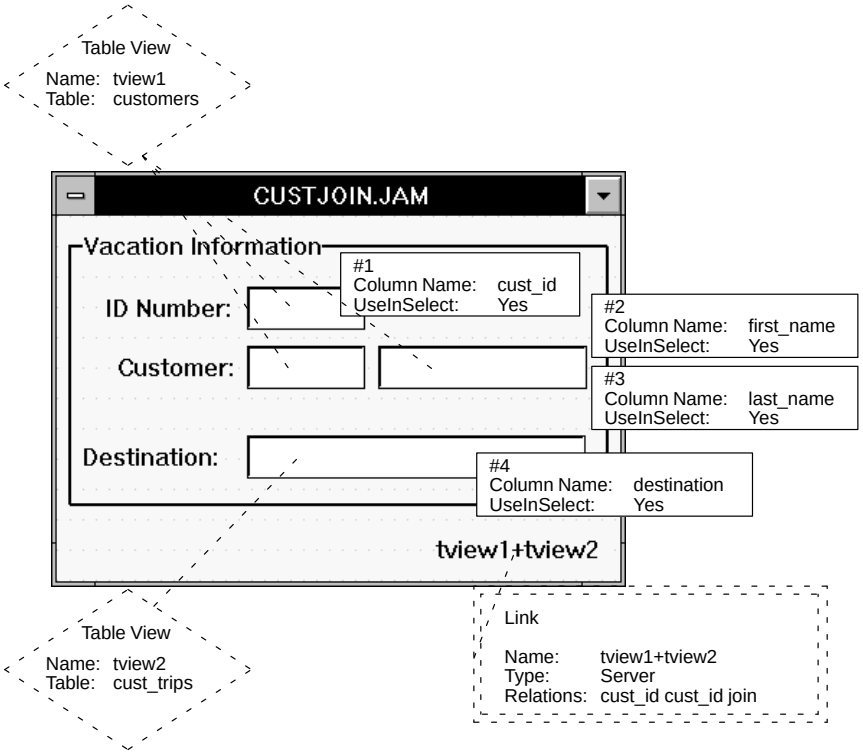
database table, then the SQL generator will automatically supply table alias names as needed. Thus, self–joins can be generated automatically.

Example 11

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
```

1. Create three widgets that are members of a table view associated with the table `customers`: `cust_id`, `first_name`, and `last_name`.
2. Create a widget to be in a table view associated with the table `cust_trips`: `destination`.
3. Create a link between the two table views with the link type as Server.



By setting the Link Type to Server, a single `SELECT` statement is generated to populate both the parent and child table views. The Relations property sets `cust_id` in the parent table view is to be joined with `cust_id` in the child table view. The Relations property value is displayed as `cust_id cust_id join`.

Generating Multiple Statements

When the Link Type is specified as Sequential, JAM 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:

widget-data-in-parent-table-view `= child_table.child_column`

Therefore, the link's Transaction⇒Relations property must specify both a column in the child's table, and a widget or column in the parent's table view.

Example 12

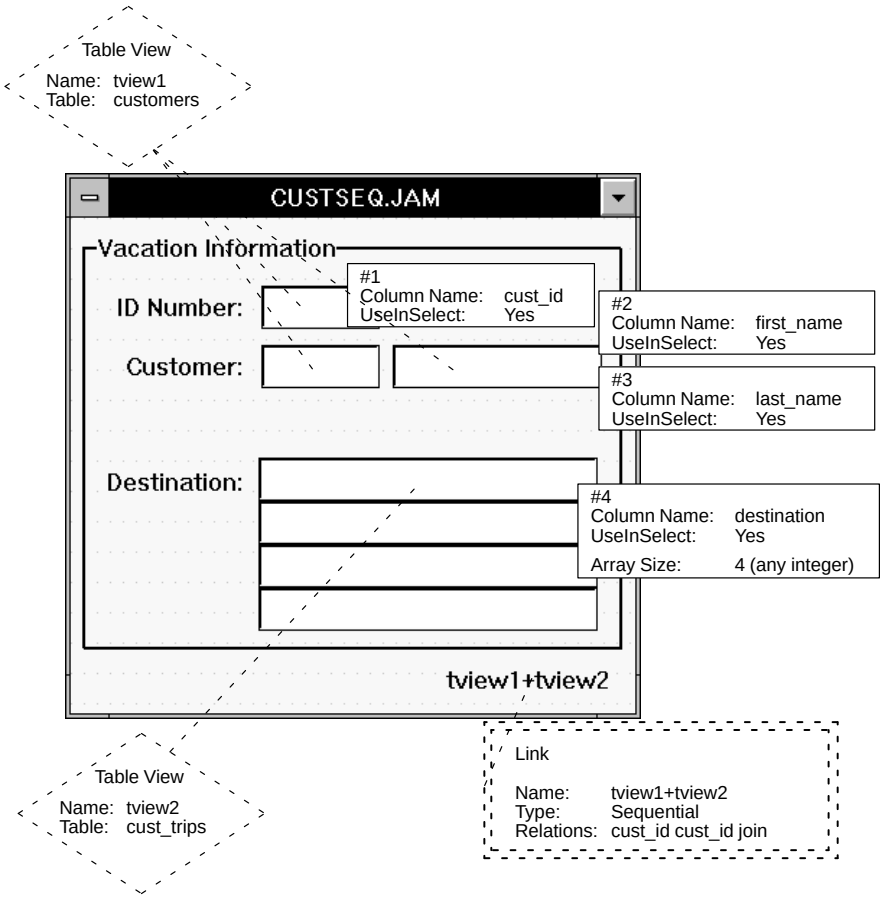
This example uses the same widgets as the previous example, but this time the `destination` widget is an array in addition to the link type being Sequential. Both examples list the customer's name and trip destination. The desired SQL is:

```
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
```

1. Create three widgets that are members of a table view associated with the table `customers`: `cust_id`, `first_name` and `last_name`.

2. Create one widget to be in a table view associated with the table `cust_trips`: `destination`.

3. Create a link between the two table views with the link type as Sequential.



Specifying a Widget in the Relations Property

If the column specified in the Parent section of the Relations property corresponds to more than one widget in the table view, the widget name must be used in the Relations property instead of the column name, with the somewhat unusual notation:

`::`*widget-name*`[+0]`

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 Updatable table views.

If a screen contains more than one table view, the link property Insert Order determines whether the statement for the parent table view or the child table view is generated first.

The following `INSERT` statement is followed by a list detailing how 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 hook function.

```
INSERT INTO table-name [ (column-list) ]
    VALUES (value-list)
```

SQL Element	Property Settings
table-name	For the table view, the value in the Database⇒Table property.
column-list	For each widget in the table view, the value in the Database⇒ColumnName property. The Use In Insert property must also be set to Yes.
value-list	Contains a value for each column in the column list taken from the current widget data. If a Use In Insert⇒Expression is provided, then it is used in place of the data in the widget. If the data in a widget is null, then JAM supplies an appropriate representation of null for the database.
Subqueries	Use hook function.

Setting the Table Name

The *table-name* is derived from the Database⇒Table property of the table view.

Setting the Column List

The *column-list* determines which 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 the column listed in the Column Name property. The Column Name property cannot be blank.

Setting the Value List

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 the widget's Use

In Insert⇒Expression property is set. The Expression property overrides the data entered in the widget.

Example 13

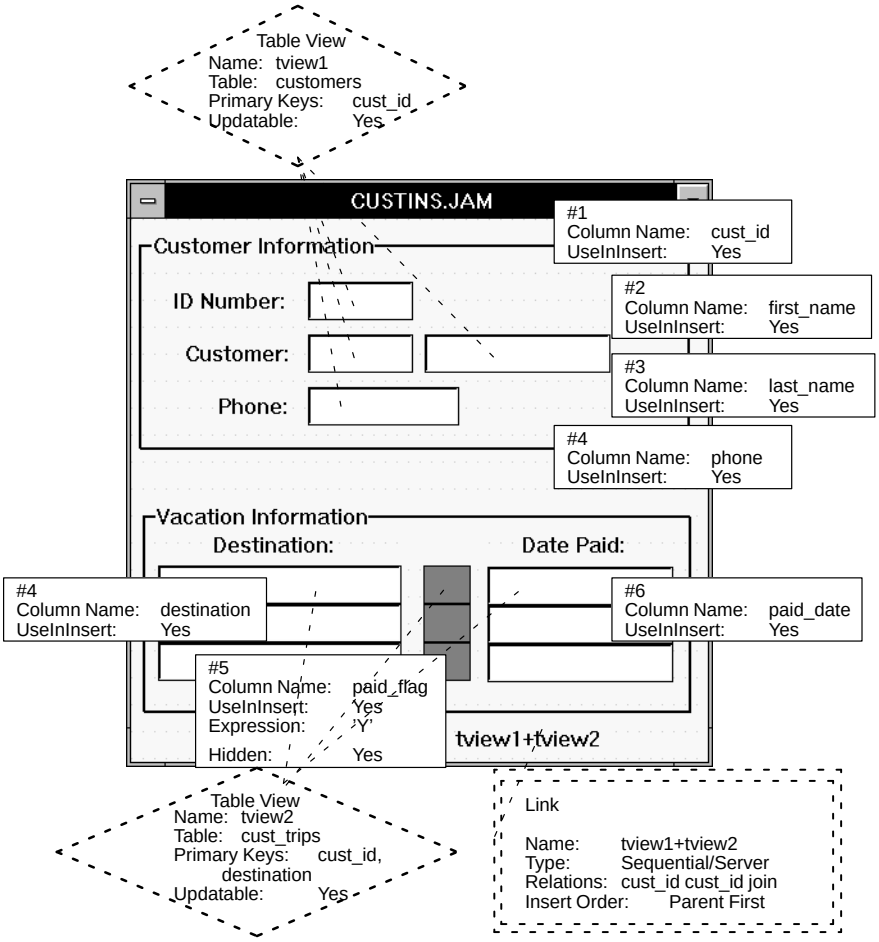
Insert values into the `customers` and `cust_trips` tables. To illustrate insert expressions, `paid_flag` will always be entered as `Y`. The desired SQL is:

```
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)
```

1. Create four widgets to be in a table view associated with the table `customers`. Since data will be inserted into the database table, the table view must be Updatable and the widgets corresponding to the Primary Keys must be onscreen.

2. Create three widgets and make them members of a table view associated with the table `cust_trips`. Since data will be inserted into the database table, the table view must be Updatable and the widgets corresponding to the Primary Keys must be onscreen.

3. Create a link between the two table views.



Implementing Optimistic Database Locking

If you choose not to use the engine's strategies for database locking, you can implement optimistic locking using version columns. First, create a version column in your database table that is one of the following data types: integer, float, or character string. Then, set the Version Column property to Yes for the corresponding widget.

In SQL `INSERT` statements, if the widget's Version Column is set to Yes, the database column corresponding to that widget is added to the column list and to the `VALUES` clause. In the `INSERT` statement, the column value is automatically set to 1. The widget designated as the version column must also have the Use In Insert property set to Yes and the C Type property set to either Int, Float, Double or Char String.

If used in the transaction manager, the default class setting for the version column is `updatable` and the styles corresponding to this class are applied.

```
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) );
```

Example 14

Insert values into the `customers` table. The desired SQL is:

```
INSERT INTO customers
   (cust_id, first_name, last_name, phone, version)
   VALUES (cust_id, first_name, last_name, phone, 1)
```

Object and Property	Value
Widget 5	
Identity⇒C Type	Int
Database⇒Column	`version`
Database⇒Version Column	Yes
Database⇒Use In Insert	Yes

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 property Update Order determines whether the statement for the parent table view or the child table view is generated first.

The following `UPDATE` statement is followed by a list detailing how 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 hook function.

```
UPDATE table-name SET column-name = value [ , ... ]
    WHERE primary-key = before-image-data
```

SQL Element	Property Settings
table-name	For the table view, the value in the Database⇒Table property.
column-name	For every widget in the table view, the value in the Database⇒Column property. The Use In Update property must be set to Yes.
value	The current widget data. If a Use In Update⇒Expression is provided, then it is used in place of the data in the widget. If the data in a widget is null, then JAM supplies an appropriate representation of null for the database.
`WHERE` clause	The columns specified in the table view's Database⇒Primary Keys property.
before-image-data	The data in the widgets which corresponded to the primary keys of the table before changes were made. This may not be the values currently stored in the widget.

Setting the Table Name

The *table-name* is derived from the Database⇒Table property of the table view.

Specifying the SET Clause

The columns listed in the `SET` clause are derived from all of the widgets in the table view whose Use In Update property is set to Yes. The *column-name* is derived from the widget's Column Name. The *new-value* is the value currently in the widget, unless Use In Update⇒Expression property is set. If the Expression is set, it overrides the value in the widget.

Setting the Primary Keys

The *primary-key* is derived from the Database⇒Primary Keys property of the table view. Each primary key column listed in the property is included in the `WHERE` clause.

Example 15

Update the phone number for a customer. The desired SQL is:

```
UPDATE customers SET phone = new_phone,
    WHERE cust_id = cust_id
```

Create four widgets and make them members of a table view associated with the table `customers`. Widget 3, with Use In Select set to Yes, displays the current phone number. Widget 4 using the select expression suggests a new phone number for the customer. Widget 4 also has the Use In Update property set to Yes so it is the value in this widget that gets written to the database.

Object and Property	Value
Widget 1	
Database⇒Column	`cust_id`
Database⇒Use In Select	Yes
Widget 2	
Database⇒Column	`phone`
Database⇒Use In Select	Yes
Widget 4	
Database⇒Column	`new_phone`
Database⇒Use In Update	Yes
Table View 1	
Database⇒Table	`customers`
Database⇒Primary Keys	`cust_id`
Transaction⇒Updatable	Yes

Implementing Optimistic Database Locking

If you choose not to use the engine's strategies for database locking, you can implement optimistic locking using version columns. First, create a version column in your database table that is one of the following data types: integer, float, or character string. Then, set the Version Column property to Yes for the corresponding widget.

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 is automatically incremented by 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 should no longer match the database value and the statement fails.

The widget designated as the version column must also have the Use In Update property set to Yes and the C Type property set to either Int, Float or Char String.

If used in the transaction manager, the default class setting for the version column is updatable and the styles corresponding to this class are applied.

```
UPDATE table-name SET column-name = value [ , ... ],
    version-column = before-image-value + 1
    WHERE primary-key = before-image-value
    AND version-column = before-image-value
```

For the example, a version column has been added to the customers table:

```
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) );
```

Example 16

Update values in the customers table. The desired SQL is:

```
UPDATE customers
    SET first_name = first_name, last_name = last_name,
    phone = phone
    WHERE cust_id = cust_id AND version = version
```

Setting In Update Where

Another method of optimistic locking would be 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.

Note that if this method is used, the Version Column property for the widget must be set to No.

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 property Delete Order determines whether the statement for the parent or child table view is generated first.

The following `DELETE` statement is followed by a list detailing how 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 hook function.

`DELETE FROM` *table-name* `WHERE` *primary-key* `=` *before-image-data*

SQL Element	Property Settings
table-name	For the table view, the value in the Database⇒Table property.
`WHERE` clause	The columns specified in the table view Primary Keys property (Database category).
before-image-data	The data in the widgets corresponding to the primary keys of the table before changes were made. Note that this may or may not be the values which are currently displayed in the widget.

Implementing Optimistic Database Locking

If you choose not to use the engine's strategies for database locking, you can implement optimistic locking using version columns. First, create a version column in your database table that is one of the following data types: integer, float, or character string. Then, set the Version Column property to Yes for the corresponding widget.

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 is added to the `WHERE` clause. Therefore, if someone else has updated or deleted the row, the version column in the `WHERE` clause should no longer match the database value and the statement fails.

The widget designated as the version column must also have the Use In Delete property set to Yes and the C Type property set to either Int, Float or Char String.

```
DELETE FROM table-name
    WHERE primary-key = before-image-value
    AND version-column = before-image-value
```

For the example, a version column has been added to the `customers` table:

```
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) );
```

Example 17

Delete values from the `customers` table. The desired SQL is:

```
DELETE FROM customers
   WHERE cust_id = cust_id AND version = version
```

Object and Property	Value
Widget 5	
Identity⇒C Type	Int
Database⇒Column	`version`
Database⇒Version Column	Yes
Database⇒Use In Select	Yes

Setting In Delete Where

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.

Note that if this method is used, the Version Column property for the widget must be set to No.

Viewing the SQL Statements

You can view the statements made by the SQL generator by:

- Using the debugger. This option provides the greatest flexibility. You can even send the generated SQL statements to a log file.
- Selecting the Trace On option from the Database menu in test and application modes. This option is less flexible, but is quicker and is often sufficient.

Examples

The following examples list the sample SQL found in this chapter and the actual SQL from the SQL generator. These statements were prepared for JDB, JYACC's prototyping database, 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 screen.

Example 2

```
SELECT destination, travel_costs, hotel, meals,
   travel_costs+hotel+meals
   FROM vacations
   WHERE destination = destination
```

The SQL generator first 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.

Example 13

```
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 (like, :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 the Use In 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 enters a customer's new phone number.

Example 15

Update the phone number for a customer. 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]
```

Modifying the SQL Statements

The automatically generated SQL statements may need additional modifications that cannot be set with the widget, table view or link properties. For additional modifications, you can write a transaction hook function to provide the desired SQL. For SQL `SELECT` statements, you can also use one of the C functions JAM provides to modify the SQL. The functions include:

- `dm_gen_change_execute_using` — Add or replace a bind value in a `DBMS EXECUTE` statement.
- `dm_gen_change_select_from` — Edit the `FROM` clause in a `SELECT` statement.
- `dm_gen_change_select_group_by` — Edit the `GROUP BY` clause in a `SELECT` statement.
- `dm_gen_change_select_having` — Edit the `HAVING` clause in a `SELECT` statement.
- `dm_gen_change_select_list` — Edit the select list in a `SELECT` statement.
- `dm_gen_change_select_order_by` — Edit the `ORDER BY` clause in a `SELECT` statement.
- `dm_gen_change_select_suffix` — Append text to the end of a `SELECT` statement.
- `dm_gen_change_select_where` — Edit the `WHERE` clause in a `SELECT` statement.

For more information on each function, refer to the *Language Reference*. For more information on writing transaction hook functions, refer to Chapter 22.

SECTION FIVE

The Transaction Manager

Introduction to the Transaction Manager

This section contains the following documentation for using the transaction manager:

- Transaction Manager Basics (Chapter 20) — Explains the process used in building a sample screen and then goes through a sample session.
- Transaction Manager Components (Chapter 21) — Contains a detailed explanation of each of the transaction manager components, like modes, table views, and links.
- Customizing Transaction Manager (Chapter 22) — Explains some of the modifications you might make to transaction manager processing, including writing your own hook functions.
- Transaction Manager Commands (Chapter 23) — Reference guide to each of the transaction manager commands listing the transaction events for each command and explaining the processing that occurs with each event.
- Transaction Manager Troubleshooting (Chapter 24) — Guidelines for building transaction manager screens as well as explanations of transaction manager errors.

The easiest way to make a screen that uses the transaction manager is to use the screen wizard which is available in the screen editor. For information about the screen wizard, refer to Chapter 5 in the *Editors Guide*.

Transaction Manager Basics

This chapter introduces basic transaction manager processing. The first section describes the process used to build an application screen. It explains how objects were imported from the database and then copied from the repository. It defines the widgets, table views and links that make up this screen. Then, the second section goes step-by-step through a sample session with the transaction manager giving an overview of the available commands and explaining how the transaction modes and styles can effect the availability of widgets and commands.

To help explain the concepts in this chapter, we are going to look at a sample screen which is based on the `videobiz` database. In this screen, you can enter a video title by name or identification code and view the actors appearing in that video and the role that each actor played. You can also enter a new video with the corresponding actors and roles. A picture of the screen appears in Figure 16.

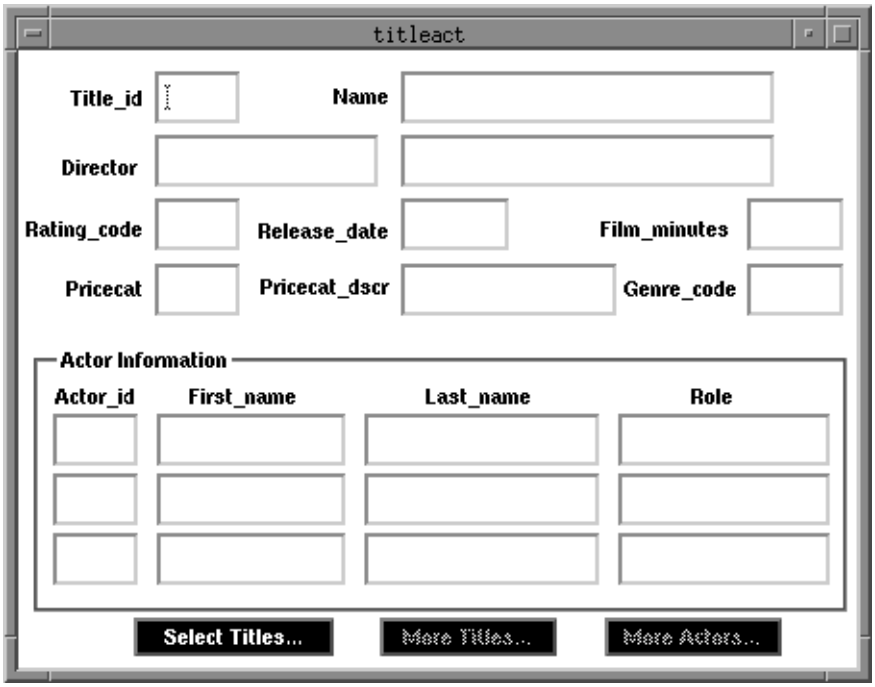


Figure 16. Sample screen which will be used to explain transaction manager processing.

Alternatively, you can use the screen wizard to build an application screen which uses the transaction manager. The screen wizard is available with the File⇒New option in the screen editor. For information about using the wizard, refer to Chapter 5 in the *Editors Guide*.

Building an Application Screen

Building the sample screen was easily accomplished using the database importer and the visual object repository. First, the `videobiz` database was created in JDB and imported into the repository. The database importer created a repository entry for each database table. Since the name of the repository entry corresponds to the database table name, the entries can be easily identified.

Each repository entry contains:

- A widget corresponding to each database column. One of the properties for the widget is the column name. Other widget properties are also set based on the column's data type.

- A label for each database column.
- A table view containing database table information.
- Links based on any foreign key definitions for the database table.

Widgets, table views and links were then copied from the repository to the application screen. 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.

Copying Repository Objects

For the sample screen, the following tables list the objects that were copied from the repository.

Table 21. Objects copied from the titles repository entry.

Repository Entry	Type of Widget	Name
titles	Text	title_id, name, genre_code, dir_last_name, dir_first_name, film_minutes, rating_code, release_date, pricecat
	Labels	LTitle_id, LName, LGenre_code, LDir_last_name, LDir_first_name, LFilm_minutes, LRating_code, LRelease_date, LPricecat
	Table View	titles (copied automatically with the text widgets)
	Link	K1titles (pricecats+titles)

Table 22. Objects copied from the roles repository entry.

Repository Entry	Type of Widget	Name
roles	Text	actor_id, role
	Labels	LActor_id, LRole
	Table View	roles (copied automatically with the text widgets)
	Link	K1roles (titles+roles), K2roles (actors+roles)

Table 23. Objects copied from the actors repository entry.

Repository Entry	Type of Widget	Name
actors	Text	first_name, last_name
	Labels	LFirst_name, LLast_name
	Table View	actors (copied automatically with the text widgets)

Table 24. Objects copied from the pricecats repository entry.

Repository Entry	Type of Widget	Name
pricecats	Text	pricecat_dscr
	Labels	LPricecat_dscr
	Table View	pricecats (copied automatically with the text widgets)

Sequence for Copying Objects

When you are creating a screen, the order used to copy objects from the repository can be important. If a screen contains multiple table views, copy the information for the table views which will be the major table views in the screen first. This ensures that any primary key widgets copied to the screen will be in the major, or *parent*, table view.

Since the focus of our sample screen is information about each video title, widgets and links from the `titles` repository entry were copied first. Then, the actor information was added to the screen.

The database stores the actor information in two different tables, `actors` and `roles`. Since the `roles` table contains a `title_id` column which provides the necessary link to the `titles` table view, information from that repository entry was copied next. Note that it was not necessary to copy the `title_id` widget itself from the `roles` entry; the transaction manager will use the `title_id` widget in the `titles` table view for SQL generation. Just the `actor_id` and the `role` widgets were copied to the screen—`actor_id` because it is part of the primary key. Since `actor_id` was already onscreen, all that was needed from the `actors` entry was `first_name` and `last_name`.

Finally, since the price category codes are not self-explanatory, `pricecat_dscr` was copied from `pricecats` to provide better descriptions.

Table Views and Links

The table views and links that were copied from the repository are JAM objects needed by the transaction manager to perform its processing. The following sections present a basic description of table views and links. If you need additional information, refer to page 354.

Table Views

A table view is a group of related widgets, generally belonging to the same database table. If a widget is a member of a table view in the repository, JAM automatically adds the widget to a table view of the same name in the destination. If the table view does not exist, JAM creates it using the properties of the table view in the repository. Thus, most table views are created automatically by the database importer and then copied from the repository as the widgets are copied.

Although the members of a table view generally belong to the same database table, this is not always the case. If a widget contains a derived value, perhaps from a database calculation or aggregate function, the widget can be added to a table view even through it does not correspond to a database column.

The sample screen contains four table views:

- `actors`
- `pricecats`
- `roles`
- `titles`

The purpose of the sample screen is to enter information about a video title and assign the actors in the database that appear in that video. The Updatable property, which determines whether data in the corresponding table can be updated, was set to No for the `actors` and `pricecats` table views since the actors and price categories should already be entered in the database before using this screen.

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 which columns or widgets connect the two table views, list the type of link—server or sequential, and list which table view is designated as the parent and which table view is designated as the child.

Setting the Parent and Child Table Views

Designating the parent and child table views helps determine the root table view and the order of processing for the 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 our sample screen, some of the Parent and Child properties had to be edited. Since the purpose of the screen is to display information about a video title, `titles` needs to be the root table view and therefore 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.

For the `K2roles` link, `roles` became the Parent, and `actors` the Child. Otherwise, the transaction manager could not determine the root table view.

Note: *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.*

Setting the Link Type

There are two types of links—sequential and server. The link type determines how `SELECT` statements are performed. In the case of a sequential link, the transaction manager executes a `SELECT` statement for each table view, generating the statement for the parent table view first. In the case of a server view, the transaction manager executes a single `SELECT` statement for the parent and child table views using a database join operation.

In the sample screen, there is a sequential link between the `titles` and `roles` table views. There is a server link between the `titles` and `pricecats` table views and another server link between the `roles` and `actors` table views.

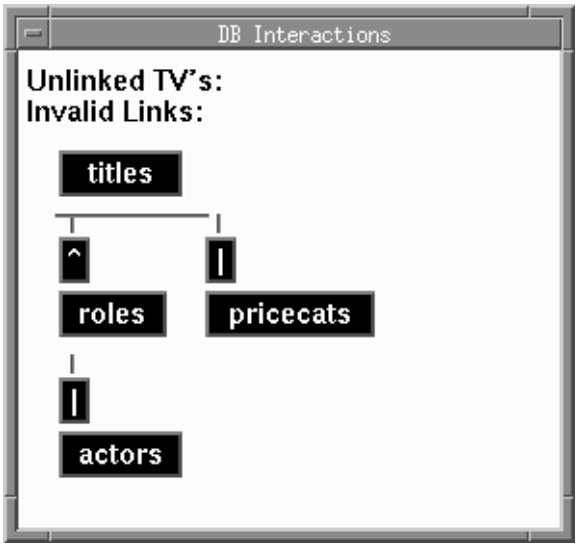


Figure 17. DB Interactions screen for the sample screen showing the linked table views and the link type.

You can view the table views and links for a screen using the DB Interactions screen. On this screen, a ^ (caret) designates a sequential link, and a | (pipe) designates a server link.

The link type only determines how `SELECT` statements are processed. Other link properties determine the order of other SQL statements. Generally, for `INSERT` and `UPDATE` statements, the statement for the parent table view is generated first. For `DELETE` statements, the statement for the child table view is generated first

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 screen is to provide information about each video title, `titles` is the root table view for our sample screen.

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.

Tree Traversal

The DB Interactions screen also graphically illustrates the table view tree that the transaction manager uses to perform its processing. 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.

A server view is either a single table view or a group of table views connected with server links. In our sample screen, there are two server views:

- `titles` (which includes the `pricecats` table view)
- `roles` (which includes the `actors` table view)

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 to that link.

The sample screen has a validation link to check the entry for a price category. The Validation Link property on the `pricecat` widget specifies the link between the `titles` and `pricecats` table views. When a new video title is entered and a valid price category is entered in `pricecat`, the description of that category is displayed in the `pricecat_dscr` widget.

Validation link processing is only performed in new and update modes, as part of the `NEW`, `COPY` or `SELECT` commands, when you are entering or updating data. Otherwise, the data in a validation link is displayed using a SQL `SELECT` statement. Since the data in the child table view should already be entered in the database, the child table view in a validation link should be non-updatable.

There is also a validation link entered for the `actor_id` widget to the `actors` table view. If you enter a valid actor identification code, the actor's name is automatically displayed.

Editing the 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 for a command or change the SQL generation performed for commands.

For our sample screen, properties were changed for the `title_id` and `name` widgets. For `title_id`, the Use In Where property was set to Yes and Operator was set to `=`. For `name`, the Use In Where property was set to Yes and Operator was set to `like%`. Before selecting data, enter an identification code or a part of a video title. Then, when you choose the `SELECT` or `VIEW` commands, the transaction manager will display the desired information. For more information on setting properties for SQL generation, refer to Chapter 18.

Push buttons were also added to the screen to perform the transaction manager commands. For information on specifying transaction manager commands, refer to page 329.

Using the Transaction Manager

Once the application screen exists with its widgets, table views and links properly defined, the screen is ready to use. Let's follow the transaction manager through a session in test mode using the sample 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.

Defining the Menu Options

The screen is displayed with the specified menu. If a menu has not been specified, JAM provides a default menu for prototyping and development. Your production application will use its own menus, push buttons, and tool bars to call transaction manager.

Database Menu

The options on the Database menu primarily handle your database connection.

Option	Description
Connect	Displays the options needed to declare a connection to the specified database engine. Enter the necessary information and press OK.
Disconnect	Closes the connection to the database.
Trace On	Activates the trace setting which displays each database driver statement in a screen window.
Trace Off	Deactivates the trace setting.
Set Connect	Select the default connection if the database engine allows multiple connections.

In addition to using the Trace On option to view the statements that are issued for each command, you can also use the debugger to view SQL generation and transaction manager event processing.

Transaction Menu

The commands listed on the Transaction menu are the basic commands needed to run the transaction manager. Once a connection to the database is established, you

can execute any of the transaction manager commands. You can also choose any of the following commands from the Transaction menu:

Option	Description
View	Retrieves one or more rows from the database for viewing purposes only.
Select	Retrieves one or more rows for possible updates.
Continue	Fetches the next group of rows if a previous `VIEW`, `SELECT`, or `CONTINUE` did not return all rows.
New	Clears the screen so the user can enter new data.
Copy	Copies the data currently on the screen, allowing the user to change it in order to enter a new row.
Save	Generates the statements necessary to update the database with the new or edited information.
Close	Aborts the current processing.
Clear	Clears the data from the screen.

It is important to note that the `NEW` command only prepares the screen for data entry. It does not insert information into the database. You must execute `SAVE` after data is entered on the screen to complete the insert. Similarly, you must also execute `SAVE` for the database to perform any updates to data retrieved with the `SELECT` command.

The commands on the Transaction menu are called using the `sm_tm_command` function. Only some of the available commands are listed on the Transaction menu. For a complete listing of the commands and options available with `sm_tm_command`, refer to Chapter 23 in the *Application Development Guide*.

Transaction Modes

The transaction manager has a series of transaction modes which determine the availability of each command. When JAM displays the default Transaction menu, some of the commands are greyed out, indicating that they are not available. This

is because the current mode precludes using those commands. The modes set by the transaction manager, along with the commands that set those modes, are:

Mode	Description	Command Selection
initial	Indicates that no processing is in progress.	`START` and `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`

When you execute a command, the transaction manager checks if the command is available in the current mode and changes to the transaction mode. If the command is unavailable in the current mode, the transaction manager reports an error. Each screen may have its own mode at any given time.

When you open a screen, the transaction manager automatically issues a `START` command which puts the screen into initial mode. For more information on the interaction of transaction commands and modes, refer to page 346.

Establishing a Connection

Once the screen is open, select the Connect option on the Database menu to establish a database connection. On the first screen, Choose Engine, you can select a database engine and a connection name. After you press OK, the options for a particular engine are listed. For JDB, a file list is displayed allowing you to select the file containing the JDB database.

After choosing the engine options, the database connection is established using the `DBMS` commands `ENGINE`, `DECLARE CONNECTION`, and `CONNECTION`. In the following example, the `ENGINE` command makes JDB the default engine. Then, the `DECLARE CONNECTION` command establishes a connection to the default engine using the options entered on the connection screen. For JDB, the only available option is `DATABASE`. Other database engines may have different options. Finally, the `CONNECTION` command sets the default connection.

In a production application, you will provide a JPL procedure or C function to open the database connection. For example:

```
DBMS ENGINE jdb
DBMS DECLARE dm_jdb1_conn CONNECTION FOR DATABASE ”videobiz”
DBMS CONNECTION dm_jdb1_conn
```

Displaying Data in the Transaction Manager

Earlier sections mentioned the two commands in the transaction manager which are used to display data from an existing database, `VIEW` and `SELECT`. Once a connection is established to the database, the commands can be invoked. You may recall that `VIEW` is used only to display information; `SELECT` is used when you allow the user to modify the selected data. For our sample session, let's use the `SELECT` command.

When the `SELECT` command is executed, data is fetched from the database and displayed in the appropriate widgets using the `DBMS` commands `DECLARE CURSOR`, `ALIAS`, and `EXECUTE`. The `DECLARE CURSOR` command creates a named cursor for the SQL `SELECT` statement. The `ALIAS` command maps the column name or select expression to the JAM destination variable. This allows you to have a widget name that is different from the database column name if you need it. Then, the `EXECUTE` command performs the SQL statement associated with the cursor named in the `WITH CURSOR` clause. If the SQL statement included bind parameters, the `USING` clause lists both the parameter and the widget whose onscreen value will be substituted.

```
DBMS DECLARE jdb1 CURSOR FOR SELECT ...
DBMS WITH CURSOR jdb1 ALIAS ...
DBMS WITH CURSOR jdb1 EXECUTE USING ...
```

This series of statements is performed for each server view, with a server view being a table view and all table views joined to it via a server link. Since our sample screen contains two server views, this series of statements would be executed two times. The first series would retrieve a video title and its associated price category from the `titles` and `pricecats` tables. The second series would fetch the actors appearing in that video and the name of their roles.

Since the `titles` table view has a sequential link with the `roles` table view, values in the `titles` table view (which is the parent) are used to fetch data for the child table view. The widget used to supply the value is named in the Relations property of the link. For our sample screen, the current value in `title_id` is used automatically to build a `WHERE` clause which will fetch only the actors in that video.

If the JAM targets for the select set are arrays, the first row fetched goes to the first occurrence, the second row to the second occurrence, etc.

Scrolling through the Select Set

The `CONTINUE` command 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. Usually you will create scrolling JAM widgets or a JAM grid for displaying the data. In environments where memory is limited, you may fetch only a small number of rows to the JAM may fetch only a small number of rows to the JAM application and buffer the rest in a file on disk. 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_BACK` command displays the previous set of data, and issuing a `CONTINUE_TOP` command displays the first set of data.

Note that JAM does *not* set backward scrolling via continuation files as the default since JAM 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. For more information on the Fetch Directions property, refer to page 377.

Styles

As you execute some of the transaction commands, you might notice changes in the widgets' behavior. Text fields may prevent input, menu choices may be deactivated, push buttons may be activated, etc. These changes occur because JAM has predefined style and class settings for each transaction mode.

The style and class settings give a consistent user interface to the application. Widgets available for data entry can have the same focus and protection settings. They can even have the same color. All of this can occur without having to write any source code or set any properties in the screen editor.

The definitions for the styles and classes are kept in a file named `styles.sty`. For an application, you can use the predefined styles and classes, edit the style and classes settings to new values, or define your own styles and classes.

Applying Styles

Let's look at our sample screen when you choose the `SELECT` command. The command, which retrieves rows from the database for possible edit, sets the transaction mode to update. The `title_id` and `actor_id` widgets are protected, preventing any edits to primary key fields. The `pricecat_dscr`, `first_name`, and `last_name` widgets are protected since they are in non-updatable table views. If you try to edit the values in these fields, the screen editor beeps, reminding the user that the fields are protected. The remaining fields are available to be updated.

For more information on styles and classes, refer to Chapter 18 in the *Editors Guide*.

Modifying Data in the Transaction Manager

If you execute a command that will modify data, such as NEW, SELECT, COPY or COPY_FOR_UPDATE, the transaction manager initiates before-image processing for all updatable table views. JAM'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.

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. Then, each time you execute the CONTINUE command, the transaction manager fetches the next screenful of data.

JAM 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 will ask the user if it should discard the changes. If the user chooses to discard, the transaction manager will proceed to the next command without changing the database. If the user chooses not to discard, transaction manager returns control to the screen. You may 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 field 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 JAM 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.

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 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 will delete the same occurrence in every array in the server view. You can program a delete line event by calling the function `sm_doccur`.

For deletions, the transaction models call the SQL generator to build a SQL `DELETE` statement with a `WHERE` clause built from the before image values of the primary key widgets.

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 will insert a line in each array in the server view. You can program an inert 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.

Discarding Your Changes

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.

Command Processing

The processing for each command in the transaction manager is determined by the *transaction model*. Source files for sample transaction models are distributed with JAM, one for each engine. The transaction manager translates each command into a series of transaction events. The processing for each event is listed in the model; however, you can change the processing for any event by writing a transaction hook function. For more information about writing hook functions, refer to page 384.

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.

Transaction Manager Components

This chapter describes in detail each of the components used in the transaction manager. Before referring to this chapter, you should be familiar with the transaction manager processing described in Chapter 20. The subjects in this chapter are:

- Transaction manager screen wizard
- Transaction manager commands
- Transaction events
- Transaction modes
- Styles and classes
- Repository
- Table views
- Links
- Transaction models

- Before image processing
- Hook functions

The following screen, which is based on the `videobiz` database, is used to illustrate the various concepts in this chapter. The steps performed to create this screen are outlined in Chapter 20.

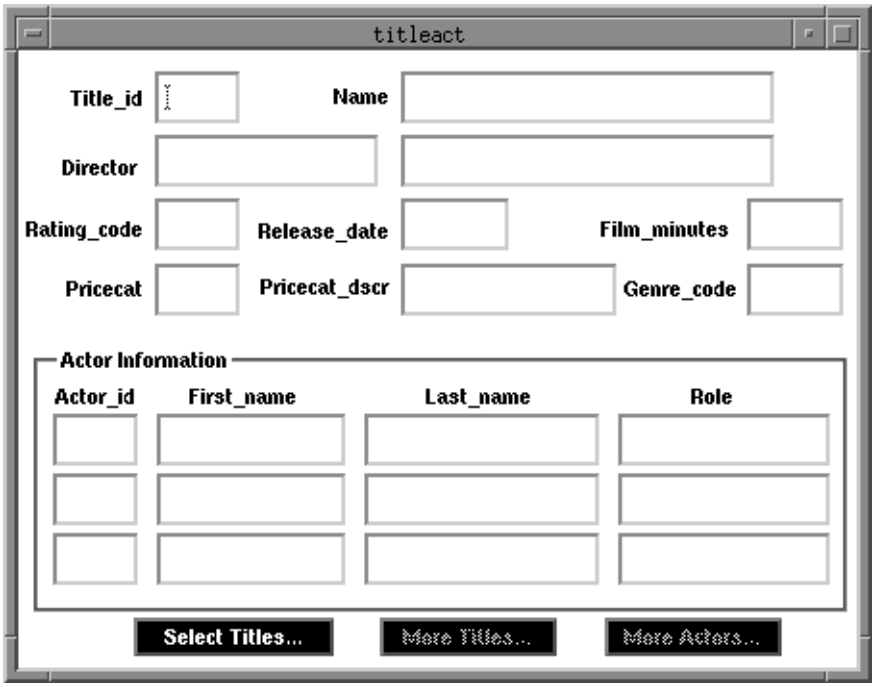


Figure 18. Sample screen which can be used to query or to enter information about video titles.

Screen Wizard

The transaction manager screen wizard is the easiest method for creating screens that use the transaction manager. The screen wizard option is available when you choose File⇒New in the screen editor.

The screen wizard can make three different types of screens—Master, Master-Detail, and Master-Detail-Subdetail—as long as you have imported your database to a repository. For more information, refer to Chapter 5 in the *Editors Guide*.

Transaction Manager Commands

A series of transaction manager commands are available for use in the transaction manager. Each command executes a different type of processing. For example, there is a command to select data from the database and a command to clear the data from the screen. When you choose a command, the transaction manager executes the transaction events associated with that command.

The default menu contains a subset of these commands for use in prototyping and development. However, you can override this menu at any time with one of your own menus, or you can use push buttons to execute commands.

When you choose a transaction manager command, you also set the transaction mode for the screen. Changing the transaction mode can also change the protection settings and display attributes of widgets on the screen. For more information about transaction modes, refer to page 346.

This section contains information applying to all transaction commands. This includes a short description of each command, information about how to specify commands, and a description of how the command is applied to the table view tree. For detailed information on each transaction manager command, including the transaction events associated with the command, refer to Chapter 23.

Description of the Commands

The commands available in the transaction manager are:

Option	Description
Change	Change to another transaction.
Clear	Clears the data from the screen.
Close	Aborts the current processing.
Continue	Fetches the next group of selected rows.
Continue_Bottom	Fetches the last set of rows using a continuation file.
Continue_Down	Fetches the group set of rows using a continuation file.
Continue_Top	Fetches the first set of rows using a continuation file.
Continue_Up	Fetches the previous set of rows using a continuation file.
Copy	Copies the data currently on the screen, allowing the user to change it in order to enter a new row.
Copy for Update	Changes the transaction mode to update, allowing the user to change the data currently on the screen.
Copy for View	Changes the transaction mode to view so the data is for viewing purposes only.
Fetch	Retrieves one or more rows for a single table view.
Force Close	Aborts the current processing.
New	Clears the screen so that the user can enter new data.
Refresh	Reapplies the styles and classes to the screen.
Save	Generates the statements necessary to update the database with the new or edited information.
Select	Retrieves one or more rows from the database for possible updates.
View	Retrieves one or more rows from the database for viewing purposes only.

It is important to note that the `NEW` command only prepares the screen for data entry. It does not insert information into the database. You must execute `SAVE` after entering data on the screen to complete the insert. Similarly, you must also execute

SAVE for the database to reflect the updates to data retrieved with the SELECT command.

Specifying Commands

Transaction manager commands are called using the function sm_tm_command. Once you specify a command, the transaction manager applies the command to each table view in the tree, unless you define table view parameters. To specify the command for specific table views, you need to include a table view name and specify the portion of the tree to be affected by the command.

For most transaction manager commands, a similar syntax is used:

sm_tm_command (*"command-name [table-view [table-view-scope]]"*)

command-name is one of the following transaction manager commands:

CLEAR	CONTINUE_DOWN	FETCH	SAVE
CLOSE	CONTINUE_TOP	FORCE_CLOSE	SELECT
CONTINUE	CONTINUE_UP	NEW	VIEW
CONTINUE_BOTTOM			

table-view, if specified, is one of the table views on the screen.

table-view-scope, an optional parameter, is generally one of the following values:

- TV_AND_BELOW which applies the command to the specified table view and all server views below it in the tree. This is the default setting if no parameter is supplied.
- BELOW_TV which applies the command to the server views below the specified table view.
- TV_ONLY which applies the command to the specified table view only.
- SV_ONLY which applies the command to the specified server view only.

Sample Command Specification

For example, the following invocation of the VIEW command without a table view or scope parameter applies this command to each linked server view, starting with the root table view:

```
sm_tm_command("VIEW")
```

To specify this command for the `titles` table only, you would need the table view name and the parameter `TV_ONLY` which limits the command to the specified table view.

```
sm_tm_command(”VIEW titles TV_ONLY”)
```

Full and Partial Commands

When a command is specified without a table view parameter, the command is applied to all linked table views. This is called a full command and the variable `TM_FULL` is set to 1. For example:

```
sm_tm_command(”VIEW”)
```

If a table view parameter is included, the command is only set to the specified portion of the tree. This is called a partial command and `TM_FULL` is set to 0. For example:

```
sm_tm_command(”VIEW titles TV_ONLY”)
```

Note that in the sample screen, the following two commands would be equivalent since the second command with the table view parameter specifies the root table view. However, `TM_FULL` is set to 1 for the first command and to 0 for the second command.

```
sm_tm_command(”VIEW”)

sm_tm_command(”VIEW titles”)
```

You can query for the current value of `TM_FULL` using `sm_tm_inquire`.

Specifying COPY

There are commands which copy data on the screen so that it can then be edited, viewed, or updated. These commands cannot be specified with table view and scope parameters. The commands are:

```
sm_tm_command (”COPY”)

sm_tm_command (”COPY_FOR_UPDATE”)

sm_tm_command (”COPY_FOR_VIEW”)
```

Specifying START

Since `START` is called automatically on screen entry, you only need to specify this command to create additional transaction manager transactions. Besides the transaction name, the syntax can also contain a table view name and a scope parameter.

`sm_tm_command (”START` *transaction [table-view [table-view-scope]]”*`)`

Specifying FINISH

`FINISH` is called automatically on screen exit and closes the current transaction manager transaction. If you have specified additional transactions with the `START`

command, you should also close them using the FINISH command. The following procedure closes two additional transactions and then changes back to the main transaction which is closed on screen exit.

```
proc close_tran
vars main_tran
main_tran = sm_tm_pinquire(TM_TRAN_NAME)
call sm_tm_command(”CHANGE tran1”)
call sm_tm_command(”FINISH”)
call sm_tm_command(”CHANGE tran2”)
call sm_tm_command(”FINISH”)
call sm_tm_command(”CHANGE :main_tran”)
return
```

Specifying CHANGE

CHANGE allows you to change to an active transaction in the transaction manager. The transaction must already exist from a call to the START command.

```
sm_tm_command (”CHANGE transaction-name”)
```

Executing Transaction Manager Commands

Each transaction manager command consists of a series of transaction events. When a command is selected, the events for that command are issued for each table view in the specified transaction tree.

The link properties determine the table view tree for the screen. To automatically participate in command processing, a table view should have at least one link to another table view in the transaction tree (or must be the only table view in the tree). When a screen is opened, the transaction manager checks the transaction tree to make sure that the root table view of the screen can be determined and that there are no circular links.

If the transaction manager reports that it cannot determine the root table view, check the Parent and Child properties of each link to see if the settings need to be reversed. If those properties are correct, set the root table view manually in the screen properties.

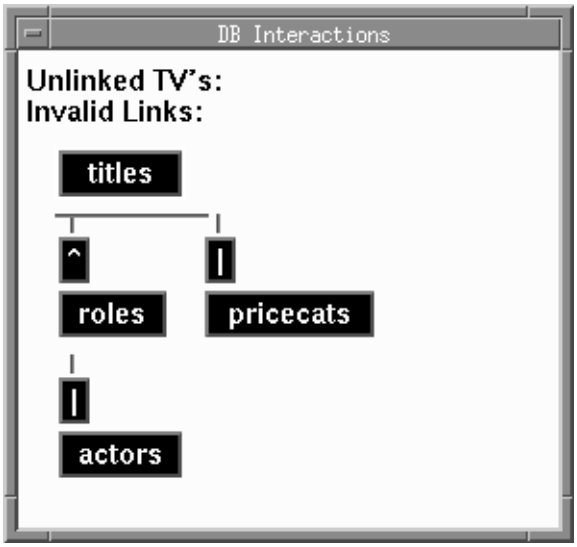


Figure 19. The DB Interactions screen presents a graphical picture of the transaction tree.

For the `VIEW` and `SELECT` commands that query information from the database, a SQL `SELECT` statement is issued for each server view in the tree. For the `SAVE` command, the SQL `INSERT`, `UPDATE`, or `DELETE` is issued for each table view.

On the sample screen, the events passed to the standard transaction models would generate the following SQL when the `VIEW` command is selected:

- A SQL `SELECT` statement joining the `titles` table (the root table view) and the `pricecats` table, which is connected to `titles` with a server link.
- A SQL `SELECT` statement joining the `roles` and `actors` tables.

A SQL join is done if the server view contains more than one table view. For the table views in a server view, the Link Type property must be Server and the Relations property must be specified as a join.

Even though you can define the portion of the table view tree in your command specification, you must understand the events associated with each command to further modify command processing. For more information about transaction events, refer to page 337.

Attaching Commands to Push Buttons

Making Push Buttons

The push buttons on our sample screen use `sm_tm_command` to query for video information, to display information about the next video or to display additional actors.

Figure 20. Push buttons on the sample screen which execute transaction manager commands.

The push button `Select Titles...` selects information about video titles using the following command string (^ is the command syntax for push buttons):

```
^sm_tm_command("SELECT")
```

The push button `More Titles...` displays information about the next video title using the following command string:

```
^sm_tm_command("CONTINUE")
```

The push button `More Actors...` fetches any additional actors and the role they played and so it needs information from just the `roles` and `actors` tables. The command string specifies the starting table view:

```
^sm_tm_command("CONTINUE roles")
```

Setting the Class Property

In the sample screen, the Class property was also set for each push button. Predefined classes already exist in JAM which can be applied to any menu item or push button. These predefined classes define whether the items assigned this class should be active or inactive in a transaction mode. If the class is marked as inactive in a mode, the menu item or push button will greyed out automatically if a command changes the screen to that mode.

For example, the Class property for the push button `Select Titles...` is `view_button`. The Class property for the push buttons `More Titles...` and `More Actors...` ,which display additional database information, are set to `continue_button`. Since the `continue_button` class is set to inactive in initial mode, the push buttons `More Titles...` and `More Actors...` are greyed out when the screen is first opened.

Table 25. The active/inactive style settings for each class and transaction mode that can be applied to push buttons and menu items.

Class	initial mode	new mode	view mode	update_occ, update mode
clear_button	active	active	active	active
close_button	inactive	active	active	active
continue_button	inactive	inactive	active	active
continue_view_button	inactive	inactive	active	inactive
copy_button	active	active	active	active
new_button	active	inactive	active	inactive
save_button	inactive	active	inactive	active
view_button	active	inactive	active	inactive

Errors

The following error conditions are associated with commands:

- The command is invalid in the current transaction mode. For more information on the modes needed for each command, refer to page 346.
- The command syntax does not allow the specified parameter. For more information on the syntax for each command, refer to Chapter 23.
- A transaction manager transaction is not in progress. Transactions are created with the START command which is called automatically on screen entry. However, screen entry calls the unnamed JPL procedure *before* it calls the START command. For this reason, transaction manager commands cannot be invoked in the unnamed procedure.

For more information on controlling error processing in the transaction manager, refer to page 371.

Transaction Events

As the transaction manager processes commands, it generates transaction events, in response to each command. Each event is "sent" to the table views involved in the transaction manager transaction. A transaction event can be thought of as a request for a table view to do some of the processing required by a command. In fact, the events generated by the transaction manager in response to commands are referred to as *requests* in order to differentiate them from other types of transaction events. Another important type of transaction event is a *slice*, which is described later.

The table view-specific processing is carried out by the transaction model associated with the table view. It is the model that knows, for example, whether or not to perform generation and execution of SQL in response to an event. Even though the processing for each event is defined and carried out by the model, it is important to note that some command processing, such as mode changes, occurs in the transaction manager itself. This processing cannot be modified directly; however, it can be affected by hook functions, by changes in the model and by additional command processing.

Traversal

When no errors occur, a request is normally sent to all table views on the screen. This allows each table view to participate in the fulfillment of each command. The order in which the table views receive the request is known as the *traversal order* of the request. Transaction manager builds a tree of table views, starting with the root table view, using the links defined on the screen. Normally, transaction manager traverses the tree, starting with the root, sending requests to each parent table view before that table view's children. However, link properties may be used to change the order when inserting, updating, or deleting records. In addition, the transaction manager commands that do table view traversal can accept an argument that specifies the initial table view.

Consider the transaction command `VIEW`. The transaction manager generates the requests `TM_PRE_VIEW`, `TM_VIEW`, and `TM_POST_VIEW`. First, `TM_PRE_VIEW` is sent to the root table view, and then to each descendent of the root table view in parent-first traversal order. The process is repeated with `TM_VIEW` and `TM_POST_VIEW`. In each of the standard transaction models supplied with JAM, SQL generation and execution takes place within processing of the `TM_VIEW` request. `TM_PRE_VIEW` and `TM_POST_VIEW` give each table view a chance to prepare prior to and after the SQL generation and execution. In each of the standard models, nothing is done in `TM_PRE_VIEW` or `TM_POST_VIEW`. However, they could be used, for example, to initiate and clear database locks that control read consistency.

Even though the processing may take place within one request, there may be several transaction events that make up that request. When a request is subdivided into further events, those events are called slices. Unlike requests, all the slices that make up a request are processed on the same table view in succession. Then, the request processing moves to the next table view. This is accomplished by using an event stack.

Event Stack

As each request is sent to the transaction model, it is pushed onto the transaction manager event stack and processed for each table view in the tree. The transaction manager processes events on the stack until the stack is empty. This means that the stacked events are processed before the next request, and therefore before the transaction manager continues with table view traversal.

Events are popped from the stack and processed in the reverse order from the order used to push them onto the stack. Thus, if events are to be processed in the order A, B, C; they must be pushed onto the stack in the order C, B, A.

There are three library functions that deal with the event 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.

In the standard transaction models, the model is responsible for slicing requests. Therefore, any hook functions can simply return `TM_PROCEED` to indicate that processing should continue and can be unaware of the next event name. Note that the hook function for an event is invoked *before* the transaction model invokes that same event.

Example

The following example appears in a similar form in all of the transaction models. In this example, 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 Chapter 23.

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 may choose to write your own function to add to the model. Be sure to track any changes you make to the transaction model since it may change in upcoming JAM 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 was accomplished by adding two events to the JDB model and calling those events after an insert.

```
#define   DUP_GEN       9901
#define   DUP_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;
```

Querying for Events

There are two library functions that deal with event names and numbers:

- `sm_tm_event` — Returns the event number associated with an event name.
- `sm_tm_event_name` — Returns the event name associated with an event number.

Summary of Events in Transaction Manager Commands

The following tables lists the typical events found in the standard transaction models for each command. They omit some specialized behavior found in a few models. However, they include a broad range of error possibilities, although encountering some of these errors might not be “typical.”

The headings in each table distinguish commands, requests, two levels of slicing, and two levels of error checking. The three events that are typically generated as a result of return values from other events, namely `TM_TEST_ERROR`, `TM_TEST_ONE_ROW` and `TM_NOTE_FAILURE`, are abbreviated in the last two columns by the first letter of the third words in their names,`E`, `O` and `F` respectively. (The other error events are not “typically” generated.)

The requests are shown in chronological order within their commands. The slices are shown in chronological order within their requests. The error checking events are done after the events that give rise to them, and before any other event processing.

For compactness, whenever it is possible, the lower level events are shown on the same line as the higher level events that give rise to them. Thus, the entry for the FETCH command compresses the information about 6 events into the following two lines:

```
Command   Request   Slice  Slice   Error  Error

FETCH     TM_FETCH      .      .      E      F
.         .         TM_SEL_CHECK     E      F
```

- The FETCH command generates only one request, TM_FETCH.
- TM_FETCH in its own right can cause a TM_TEST_ERROR event to be generated by the transaction manager (by returning TM_CHECK).
- The TM_TEST_ERROR event can cause a TM_NOTE_FAILURE event to generated by transaction manager (by returning TM_FAILURE).
- TM_FETCH also has a slice that it generates, TM_SEL_CHECK.
- TM_SEL_CHECK can cause a TM_TEST_ERROR event to be generated by transaction manager (by returning TM_CHECK).
- The TM_TEST_ERROR event can cause a TM_NOTE_FAILURE event to generated by transaction manager (by returning TM_FAILURE).

It should also be noted that, for example, the line for the TM_SEL_BUILD_PERFORM event in the CONTINUE command has two events (E and F) in its first ERROR column, because TM_SEL_BUILD_PERFORM can return TM_CHECK, in addition to TM_FAILURE and TM_OK. This line is to be read as associating the second F with the E, not with the first F. It is the TM_TEST_ERROR event from TM_CHECK that can give rise to a further TM_NOTE_FAILURE event.

Note that there are differences between the stated behavior of TM_POST_SAVE1 in the SAVE command and in the various other commands. For SAVE, there may have been a variety of database operations that will not occur for the other commands in the standard transaction models. This increases the complexity of the processing, and the number of possible events.

For a description of the processing for each event, refer to Chapter 23.

Command	Request	Slice	Slice	Error	Error
CHANGE					
CLEAR	TM_PRE_CLOSE				
.	TM_CLOSE				
.	TM_QUERY				
.	TM_DISCARD				
.	TM_POST_CLOSE	TM_POST_SAVE1			
.	.	.	TM_POST_SAVE2	F	
.	.	TM_POST_SAVE2		F	
.	TM_PRE_CLEAR				
.	TM_CLEAR				
.	TM_POST_CLEAR				
CLOSE	TM_PRE_CLOSE				
.	TM_CLOSE				
.	TM_QUERY				
.	TM_DISCARD				
.	TM_POST_CLOSE	TM_POST_SAVE1			
.	.	.	TM_POST_SAVE2	F	
.	.	TM_POST_SAVE2		F	
CONTINUE	TM_FETCH	.	.	E	F
.	.	TM_SEL_CHECK.		E	F
.	TM_PRE_SELECT				
.	TM_SELECT	TM_GET_SEL_CURSOR		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_SELECT				
.	TM_PRE_VIEW				
.	TM_VIEW	TM_GET_SEL_CURSOR		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_VIEW				
CONTINUE_BOTTOM	TM_CONTINUE_BOTTOM			E,F	F
.	.	TM_SEL_CHECK.		E	F
.	TM_PRE_SELECT				
.	TM_SELECT	TM_GET_SEL_CURSOR		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_SELECT				
.	TM_PRE_VIEW				
.	TM_VIEW	TM_GET_SEL_CURSOR		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_VIEW				

Figure 21. `CHANGE`, `CLEAR`, `CLOSE`, `CONTINUE`, *and* `CONTINUE_BOTTOM` *commands.*

Command	Request	Slice	Slice	Error	Error
CONTINUE_DOWN	TM_CONTINUE_DOWN	.		E,F	F
.	.	TM_SEL_CHECK.		E	F
.	TM_PRE_SELECT				
.	TM_SELECT	TM_GET_SEL_CURSOR		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_SELECT				
.	TM_PRE_VIEW				
.	TM_VIEW	TM_GET_SEL_CURSOR		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_VIEW				
CONTINUE_TOP	TM_CONTINUE_TOP	.		E,F	F
.	.	TM_SEL_CHECK.		E	F
.	TM_PRE_SELECT				
.	TM_SELECT	TM_GET_SEL_CURSOR		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_SELECT				
.	TM_PRE_VIEW				
.	TM_VIEW	TM_GET_SEL_CURSOR		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_VIEW				
CONTINUE_UP	TM_CONTINUE_UP.	.		E,F	F
.	.	TM_SEL_CHECK.		E	F
.	TM_PRE_SELECT				
.	TM_SELECT	TM_GET_SEL_CURSOR		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_SELECT				
.	TM_PRE_VIEW				
.	TM_VIEW	TM_GET_SEL_CURSOR		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_VIEW				

Figure 22. `CONTINUE_DOWN`, `CONTINUE_TOP`, *and* `CONTINUE_UP` *commands.*

Command	Request	Slice	Slice	Error	Error
COPY	TM_PRE_CLOSE				
.	TM_CLOSE				
.	TM_QUERY				
.	TM_DISCARD				
.	TM_POST_CLOSE	TM_POST_SAVE1			
.	.	.	TM_POST_SAVE2	F	
.	.	TM_POST_SAVE2		F	
.	TM_PRE_COPY				
.	TM_COPY				
.	TM_POST_COPY				
COPY_FOR_UPDATE	TM_PRE_CLOSE				
.	TM_CLOSE				
.	TM_QUERY				
.	TM_DISCARD				
.	TM_POST_CLOSE	TM_POST_SAVE1			
.	.	.	TM_POST_SAVE2	F	
.	.	TM_POST_SAVE2		F	
.	TM_PRE_COPY_FOR_UPDATE				
.	TM_COPY_FOR_UPDATE				
.	TM_POST_COPY_FOR_UPDATE				
COPY_FOR_VIEW	TM_PRE_CLOSE				
.	TM_CLOSE				
.	TM_QUERY				
.	TM_DISCARD				
.	TM_POST_CLOSE	TM_POST_SAVE1			
.	.	.	TM_POST_SAVE2	F	
.	.	TM_POST_SAVE2		F	
.	TM_PRE_COPY_FOR_VIEW				
.	TM_COPY_FOR_VIEW				
.	TM_POST_COPY_FOR_VIEW				
FETCH	TM_FETCH	.	.	E	F
.	.	TM_SEL_CHECK		E	F
FINISH	TM_FINISH	.	.	E	F
FORCE_CLOSE	TM_PRE_CLOSE				
.	TM_DISCARD				
.	TM_POST_CLOSE	TM_POST_SAVE1			
.	.	.	TM_POST_SAVE2	F	
.	.	TM_POST_SAVE2		F	
NEW	TM_PRE_CLOSE				
.	TM_CLOSE				
.	TM_QUERY				
.	TM_DISCARD				
.	TM_POST_CLOSE	TM_POST_SAVE1			
.	.	.	TM_POST_SAVE2	F	
.	.	TM_POST_SAVE2		F	
.	TM_PRE_NEW				
.	TM_NEW				
.	TM_POST_NEW				

Figure 23. `COPY`, `COPY_FOR_UPDATE`, `COPY_FOR_VIEW`, `FETCH`, `FORCE_CLOSE`, *and* `NEW` *commands.*

Command	Request	Slice	Slice	Error	Error
REFRESH					
SAVE	TM_PRE_SAVE				
.	TM_SAVE				
.	TM_DELETE	TM_GET_SAVE_CURSOR		F	
.	.	TM_DELETE_DECLARE		E,F	F
.	.	TM_DELETE_EXEC		O,F	F
.	TM_UPDATE	TM_GET_SAVE_CURSOR		F	
.	.	TM_UPDATE_DECLARE		E,F	F
.	.	TM_UPDATE_EXEC		O,F	F
.	TM_INSERT	TM_GET_SAVE_CURSOR		F	
.	.	TM_INSERT_DECLARE		E,F	F
.	.	TM_INSERT_EXEC		O,F	F
.	TM_POST_SAVE	TM_POST_SAVE1		E,F	F
.	.	.	TM_GIVE_UP_SAVE_CURSOR	E	F
.	.	.	TM_POST_SAVE2	F	
.	.	TM_POST_SAVE2			
SELECT	TM_PRE_SELECT				
.	TM_SELECT	TM_GET_SEL_CURSOR		F	
.	.	TM_PREPARE_CONTINUE		E	F
.	.	TM_SEL_GEN			F
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_SELECT				
START	TM_START				
VALIDATE_LINK	TM_PRE_VAL_LINK				
.	TM_VAL_LINK	TM_GET_SAVE_CURSOR		F	
.	.	TM_VAL_GEN			F
.	.	TM_VAL_BUILD_PERFORM		E,F	F
.	.	TM_VAL_CHECK		E	F
.	TM_POST_VAL_LINK				
VIEW	TM_PRE_VIEW				
.	TM_VIEW	TM_GET_SEL_CURSOR		F	
.	.	TM_PREPARE_CONTINUE		F	
.	.	TM_SEL_GEN		F	
.	.	TM_SEL_BUILD_PERFORM		E,F	F
.	.	TM_SEL_CHECK		E	F
.	TM_CLEAR				
.	TM_POST_VIEW				

Figure 24. REFRESH, SAVE, SELECT, START, VALIDATE_LINK, *and* VIEW *commands.*

Transaction Modes

The transaction manager has a series of transaction modes which determine the availability of each transaction command. For example, when you display the default Transaction menu, some of the commands are greyed out, indicating that they are not available because the current mode precludes using those commands. The following table lists the modes set by the transaction manager and the commands that initiate those modes:

Mode	Description	Command Selection
initial	Indicates that no processing is in progress.	`START` and `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 table lists the commands that set the listed mode. The following commands do not change the transaction mode but are only available in certain modes:

- `CONTINUE` (and its variants) and `FETCH` are only available in update or view mode.
- `SAVE` is only available in new or update mode.

The `CLEAR` command, 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 may 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. For more information on styles and classes, refer to page 349.

Command Availability

The following table lists whether a transaction command is valid in the specified mode and whether full or partial commands are allowed. Full commands operate

on the entire table view tree and do not include a table view or scope parameter. For example:

```
^sm_tm_command (”VIEW”)
```

Partial commands have a table view parameter and only operate on a portion of the tree:

```
^sm_tm_command (”VIEW roles”)
```

Note that the default Transaction menu may have some of these commands as inactive in the current mode. The table is included to illustrate what range of specification is possible for transaction manager commands.

Table 26 uses the following codes:

- Y – Valid for full or partial commands
- N – Invalid
- F – Valid for full commands; invalid for partial commands
- P – Valid for partial commands; invalid for full commands

Table 26. Listing of the transaction manager commands that are valid in each transaction mode.

Transaction Command	Initial Mode	New Mode	Update Mode	View Mode
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	F	F	F	F
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
SAVE	N	Y	Y	N
SELECT	Y	Y	Y	*
START	Y	Y	Y	Y
VIEW	Y	Y	Y	Y

* A partial SELECT in view mode is treated as a partial VIEW command; a full SELECT in view mode is treated as a SELECT command.

Styles and Classes

As you execute some of the transaction commands, you might notice changes in the widgets' behavior. Text fields may prevent input, menu choices may be de-activated, push buttons may be activated, etc. These changes occur because JAM has predefined style and class settings for each transaction mode.

The style and class settings give a consistent user interface to the application. Widgets available for data entry can have the same focus and protection settings. They can even have the same color. All of this can occur without having to write any source code or set any properties in the screen editor.

The definitions for the styles and classes are kept in a file named `styles.sty`. For an application, you can use the predefined styles and classes, edit the style and classes settings to new values, or define your own styles and classes.

Classes

JAM's predefined classes establish three categories for data in an application screen:

- Data included for informational purposes only which should not be updated.
- Data that needs to be entered into the database.
- Data that once entered into the database should not be updated, such as primary keys.

The names of these predefined classes are:

Transaction Class	Description
non_updatable	Widget is a member of a non-updatable table view.
updatable	Widget is not part of the table's primary key and is a member of an updatable table view.
primary_key	Widget is part of the table's primary key and is a member of an updatable table view.

In addition, there are other predefined classes to use with menu selections and push buttons. For more information on these classes, refer to page 335.

Each widget has a Class property. Initially, the Class property for the text widgets is set to "default," and JAM determines the widget's class from values in other

properties. Since the Class property for the widgets in our sample screen is set to "default," Figure 25 lists which class is applied to each widget.

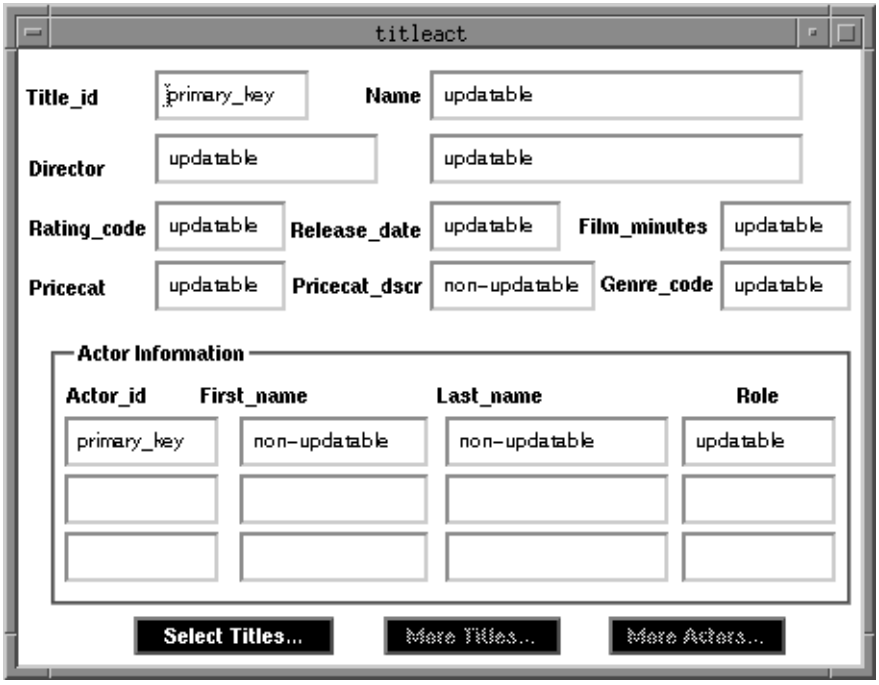


Figure 25. The sample screen with the default class setting for each data widget.

In the sample screen, the classes are assigned as follows:

- `title_id` and `actor_id` are primary keys so the class for these two widgets becomes `primary_key`.
- `first_name`, `last_name` and `pricecat_dscr` are members of table views where the Updatable property is set to No so their class is `non_updatable`.
- The class for the remaining widgets is `updatable`.

However, you can change the class of any widget by editing the Class property.

The access to widgets in each of these classes needs to change as the transaction mode changes. For example, you need to enter a value in a primary key field for an insert, but you would want to prevent updates to this field. Changes in access are determined by the style assigned to each transaction mode.

Specifying Styles

When you specify a style, you set a series of widget properties in any of the following categories: focus protection, clearing protection, input protection,

validation, keystroke filter, background color, and foreground color. The following styles are predefined:

Style	Property Settings
change	allow focus, input, clearing, and validation
edit	allow focus, input, clearing, and validation
show	prevent focus, input, and clearing allow validation
visit	allow focus and validation prevent 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 335.

Specifying Classes

When you specify a class, you assign a style to each of the transaction modes. For JAM's predefined classes, the styles assignments are listed in Table 27.

Table 27. Style settings for each of the default classes.

Transaction Class	initial	new	view	update occ	update
non_updatable	edit	show	visit	show	show
primary key	edit	edit	visit	visit	edit
updatable	edit	edit	visit	change	edit

For the predefined series of classes, the style settings accomplish the following:

Transaction Class	Description
non_updatable	For widgets in a non-updatable table view, data displayed in the widget cannot be edited in any mode.
primary_key	For widgets that are part of the table's primary key in an updatable table view, the data displayed in the widget cannot be edited in update mode or in view mode.
updatable	For widgets in an updatable table view that are not part of the table's primary key, the data can be edited except in view mode.

Note that there are two styles corresponding to update mode. When data is fetched in update mode to arrays, it is usually desirable to treat the occurrences with fetched data differently than the occurrences without fetched data. For example, JAM normally prevents input on a Primary Key widget when it contains data fetched from the database. This prevents the user from changing the value of the key. However, when there is no fetched data in an occurrence, JAM normally allows input in the Primary Key widget to order to enter new records.

To enable a different behavior for occurrences with fetched data, you associate two styles with update mode when you create a transaction class. One style is for occurrences with fetched data (referred to in the styles editor as `update_occ` mode), and one is for occurrences without fetched data (referred to in the styles editor as `update` mode).

Before fetching data, JAM applies the style corresponding to `update` to the occurrences of each widget. After fetching data into a widget, JAM applies the `update_occ` style to each occurrence containing fetched data (even if what was fetched was blank or null).

In addition, there are other predefined classes to use with menu selections and push buttons. For more information on these classes, refer to page 335.

Applying Styles and Classes

Let’s look at our sample screen when you execute the `SELECT` command. The command, which retrieves rows from the database for possible edit, sets the transaction mode to update. The `title_id` and `actor_id` widgets are protected, preventing any edits to primary key fields. The `pricecat_dscr`, `first_name`, and `last_name` widgets are protected since they are in non-updatable table views. If you try to edit the values in these fields, the screen editor beeps, reminding the user that the fields are protected. The remaining fields are available to be updated.

You can edit the predefined styles or define new styles for your application if needed. For more information on styles, refer to Chapter 18 in the *Editors Guide*.

Repository

Since the repository is used as a basis for many transaction manager screens, a description of how the repository can be used is included here.

Constructing the Database First

To build screens for use with the transaction manager, it is recommended that you begin by designing and building the database. JYACC has a series of database interfaces to JAM for many DBMS products. This includes products by several DBMS vendors as well as JYACC's prototyping database, JDB.

Once the database exists, use JAM to create a repository and import your database tables into the repository. A separate repository screen is created for each table in the database. This repository screen contains the following objects:

- Text widgets for each of the database columns.
- Labels for each widget corresponding to the database column names.
- A table view listing all the database columns as members of that table view.
- Links corresponding to the foreign key definitions.

If your database engine does not support primary and foreign key definitions, it is recommended that you modify your repository screens to contain that information. Enter the primary keys as part of the Database category in the table view properties. For any foreign key definitions, you need to create a link and enter all the link properties. For information on how to create links, refer to the *Editors Guide*.

You may wish to enhance the repository in other ways. You may want to make the link names more descriptive. You may want to create additional repository screens to contain push buttons and other widgets used throughout the application.

Once the repository exists, you can start building your application screens by copying objects from the repository screens. For the objects imported from the database, the database column and table information needed by the transaction manager already exists. However, you may wish to edit other widget properties.

Constructing the Screens First

Even though it is recommended that you start by constructing the database, it is not mandatory. You can start the development process by building screen prototypes.

Then, in order to use the transaction manager, you need to add the database information later through inheritance or by editing the properties for the widgets, table views, and links. Depending on the size of the application, this could be a large task which would be avoided by starting with the database.

Constructing Screens with the Screen Wizard

When you create screens with the screen wizard, two entries are made in your repository:

- `smwizard`
- `smwizis`

The transaction manager screen wizard is the easiest method for creating screens that use the transaction manager. The screen wizard option is available when you choose File⇒New in the screen editor.

The screen wizard can make three different types of screens—Master, Master-Detail, and Master-Detail-Subdetail—as long as you have imported your database to a repository. For more information, refer to Chapter 5 in the *Editors Guide*.

Table Views

A *table view* is a group of related widgets, generally belonging to the same database table. If a widget is a member of a table view in the repository, JAM automatically adds the widget to a table view of the same name in the destination. If the table view does not exist, JAM creates it using the properties of the table view in the repository. Thus, most table views are created automatically by the database importer and then copied from the repository to application screens.

Even though the members of a table view generally belong to the same database table, this is not always the case. If a widget contains a derived value, perhaps from a database calculation or aggregate function, the widget can be added to a table view even through it does not correspond to a database column.

An application screen can even have a blank table view without any members, even though, by definition, a table view is a group of widgets. In fact, empty table views are sometimes necessary for SQL generation.

Table View Properties

Table views have properties just like other screen objects. Some of the properties are derived from the information in the database:

- The name of the table view, generally the same as the database table.
- The name of the database table corresponding to this table view.
- The primary key (or unique index) columns for the database table.
- The columns that are in the database table. (This list is for reference only.)
- The widgets used to sort the data fetched from the database.

Some of the properties determine what processing occurs for the table view in the transaction manager:

- A setting to determine whether the database table can be updated. If you want the widgets that are a part of this table view to display information, but not allow anyone to update this information, set Updatable to No.
- The transaction hook function to be called for this table view. If specified, the hook function is called for each transaction event.
- The transaction model to use for this table view. If blank, the transaction manager uses the standard model.
- The Fetch Directions property which determines if non-sequential scrolling is allowed for the table view.

To view a table view's properties, select the table view on the DB Interactions screen or the Widget List, and then switch focus to the Properties window.

Table Views in a Repository

A table view is created automatically when you import a database table into the repository. This table view is named for the database table and is stored with the repository entry. Then, when you copy a widget from this repository entry, JAM again creates the table view, provided the widget belonged to a table view in the repository.

Table Views in an Application Screen

When you copy a widget from your repository to an application screen, JAM creates the table view for that widget, if it was a member of a table view in the repository. If you copy additional widgets from the same repository entry, those widgets will automatically be added to the correct table view in the application screen.

If you copy widgets from another repository entry, JAM creates a new table view, corresponding to the database table for those widgets.

For a widget to participate in transaction manager processing and SQL generation, the widget must be a member of a table view. All the members of a table view that participate in SQL `INSERT`, `UPDATE` and `DELETE` statements must have the same number of onscreen occurrences and the same number of maximum occurrences.

New widgets that you create on the screen can be added to the appropriate table view. Generally, widgets that you add to table views are used to perform calculations or to display information derived from the database table. For information on how to add members to a table view, refer to Chapter 21 in the *Editors Guide*.

If desired, a table view can be created manually. This is especially useful if you want to execute a self join in the SQL generation. For more information on creating table views, refer to Chapter 21 in the *Editors Guide*.

Setting the Root Table View

The *root table view* determines the base of the event processing occurring on the current screen. For example, in a master-detail screen, the master database table would be the root table view. In most cases, the transaction manager determines the root table view automatically and displays it at the top of the DB Interactions screen.

The transaction manager uses the Parent and Child properties of the links on a screen to calculate the root table view. If it is unsuccessful, the screen displays the error message “Root table view name not supplied or not valid.” If you get this message, 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 under Transaction.

Example

The sample screen contains four table views:

- `actors`
- `pricecats`
- `roles`
- `titles`

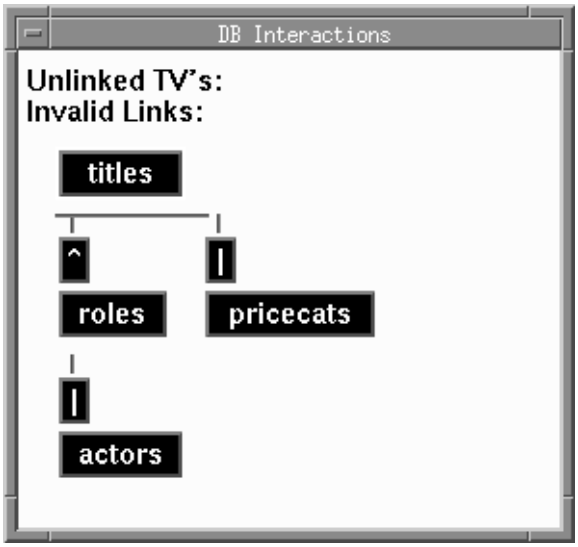


Figure 26. DB Interactions screen for the sample screen.

In the sample screen, you can enter information about a video title and assign the actors in the database that appear in that video. The Updatable property was set to No for the `actors` and `pricecats` table views since the actors and price categories should already be entered in the database before using this screen.

Since the purpose of the sample screen is to provide information about each video title, `titles` has been correctly designated as the root table view.

Links

A link defines the relationship between two table views. If a screen contains more than one table view, you need to have links joining the various table views which describes their relationship. Otherwise, the unlinked table views will not participate in any command processing.

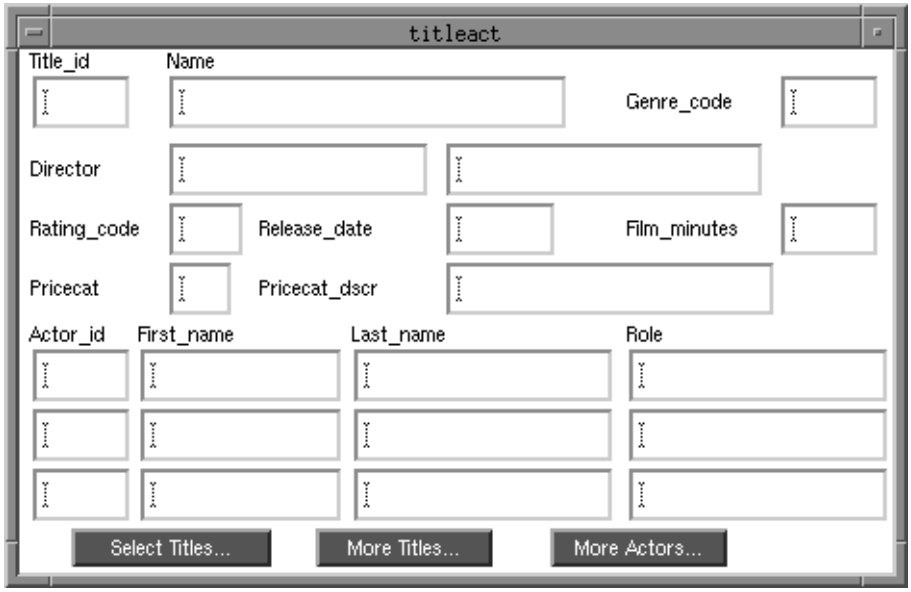
Note: *Advanced users may choose to write hook functions or JPL procedures to start transaction manager transactions for unlinked table views.*

Creating Links

When you import a database table to the repository, JAM automatically creates links corresponding to the foreign key definitions for that database table. Then, when you create an application screen, you copy the applicable links along with the other database widgets.

If a table view is not linked to any other table view, you need to create the link manually and enter the necessary properties. The parent and child table views must be defined for the link. The columns joining the two table views must be specified in the Relations property. In addition, the link type may need to be changed.

To help explain links, let's look at the sample screen:



It contains four table views:

- `titles`
- `roles`
- `actors`
- `pricecats`

Every table view is linked to at least one other table view on the screen:

- `titles+roles`
- `roles+actors`
- `titles+pricecats`

Setting the Parent and Child Properties

In a link, one table view is designated as the parent and the other table view 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 is associated with both the Parent and Child table views.

When you copy links from the repository, you might need to reverse the order of the Parent and Child properties. For example, the table view to be designated the root table view must be the Parent table view in any link in which it appears.

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.

Tree Traversal

The DB Interactions screen also graphically illustrates the table view tree that the transaction manager uses to perform its processing. When a screen is first displayed, a tree is generated using the table views and links to determine the *server views*. Each server view is either a single table view or a group of table views connected with server links. In our sample screen, there are two server views:

- `titles` (which includes `pricecats`)
- `roles` (which includes `actors`)

Setting the Link Type

There are two types of links—sequential and server. The link type determines how SQL `SELECT` statements are performed. In the case of a sequential link, the transaction manager executes a `SELECT` statement for each table view, generating the statement for the parent table view first, followed by a statement for the child table view. In the case of a server view, the transaction manager executes a single `SELECT` statement including both the parent and child table views. Server links are so named because the database server does the work, generally using a database join. Sequential links are so named because the statements are done sequentially.

In our sample screen, the link type for the `titles` and `roles` table views is sequential. One row in the `titles` table has the potential of having several associated rows in the `roles` table. Therefore, the processing for the `titles` table view is done first followed by the processing for the `roles` table view. The link type for the `roles` and `actors` table views is server. The link type for the `titles`

and `pricecats` table views is also server. There is a one-to-one relationship between `actor_id` in the `roles` table and `actor_id` in the `actors` table. The database server can perform a join to obtain the necessary information.

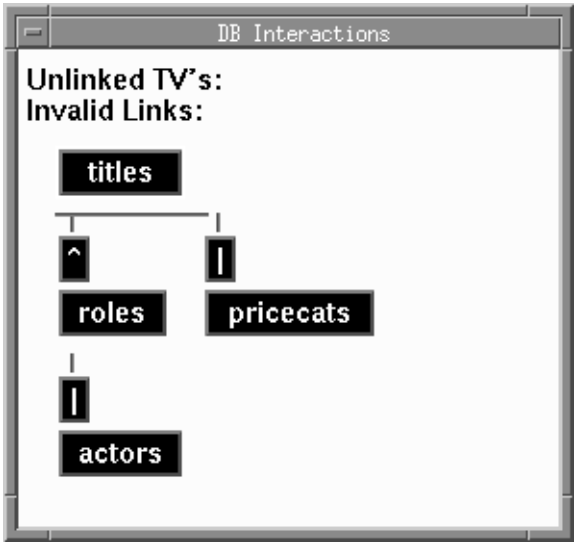


Figure 27. DB Interactions screen for the sample screen.

You can view the table views and links for a screen using the DB Interactions screen. On this screen, a ^ (caret) designates a sequential link, and a | (pipe) designates a server link.

The link type only determines how `SELECT` statements are processed. Other link properties determine the order of other SQL statements. Generally, for `INSERT` and `UPDATE` statements, the statement for the parent table view is generated first. For `DELETE` statements, the statement for the child table view is generated first

Setting the Relations Property

The Relations property specifies the columns or widgets that connect the two table views. You can also specify whether the relationship is a join or a lookup.

Using Join

Generally, the Relations Type for two table views is specified as Join. The column or columns needed to construct a SQL join for the two table views are listed in the Parent and Child columns.

Using Lookup

The Relations Type can also be specified as Lookup. 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 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. Note that the lookup relations are ignored when executing `VIEW` and `SELECT`.

The only restriction for the Lookup property is that the parent and the child table views must relate to each other directly, without any table views between them.

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. When you move to the next field, the transaction manager checks to see if it exists in the database. If the value exists, it displays data for any widgets belonging to the child table view named in the link. If the value does not exist, it displays the error `Invalid Entry`.

It is very simple to specify a validation link for a widget. Create the desired link if it does not exist. Then, in the widget's Database properties, set the Validation Link property to that link.

Validation link processing is only performed in new and update modes, as part of the `NEW`, `COPY`, `COPY_FOR_UPDATE`, or `SELECT` commands, when you are entering or updating data. Otherwise, the data in a validation link is displayed using a SQL `SELECT` statement. Since the data in the child table view should already be entered in the database, the child table view in a validation link should be non-updatable.

It should be noted that the validation link is processed *after* all other field level validation. Therefore, JAM executes the field validation function and field level JPL before it calls the validation link processing.

Also, if you are using a validation link in an array, you need to set the Link Type to be Server. If the Link Type is Sequential, SQL generation assumes a one-to-many (1:*n*) relationship instead of a one-to-one (1:1).

Example

In the sample screen, there is a validation link to check the entry for the price category. The Validation Link property on the `pricecat` widget specifies the link between the `titles` and `pricecats` table views. 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. Typically, the child table view, in this case `pricecats`, is non-updatable.

Figure 28. The validation link on the pricecat widget checks for a valid entry and displays information in the pricecat_dscr widget.

Enforcing Foreign Keys with Validation Links

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. No application should enter a value for `pricecat` in the `titles` table unless the value already exists in the `pricecats` table.

A JAM validation link enforces a foreign key integrity constraint. When a JAM link is to used to enforce a foreign key, the parent table view ”references” the child table view. The parent table view will be updatable; the child table view must be non–updatable.

A JAM validation link performs the following steps by default:

- Validates a foreign key integrity constraint for the current field in the updatable table view.
- Selects columns from the referenced table to the non–updatable child table view.

The first step verifies that the value in the field with the validation link is valid. To test it, it generates and executes the following statement

```
dbms declare cursor cursor for select 1 from pricecats
   where ((pricecats.pricecat == ::l0))
dbms with cursor cursor alias @null
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 transaction manager knows the value is valid. Transaction manager deliberately avoids selecting any data to the application. If transaction manager attempted to fetch to the `pricecat` widget, the field would be cleared when no data was found. By fetching to `@null`, transaction manager avoids overwriting the user’s data entry.

If the first select is successful, transaction manager generates and executes a second select to populate any fields in 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, JAM displays the error `Invalid Entry`.

Adding a Lookup to a Validation Link

A validation link may also perform the following optional step:

- Selects columns from the referenced table to fields in the updatable table view.

This allows the validation link to supply some suggested values for other fields in the updatable table view.

For example, consider a change to the definition of the `titles` table:

```
create table titles (
   title_id  int not null,
   title     char(20),
   genre_codechar(5),
   ...
   pricecat  char(1),
   preview_days  int,
   primary key (title_id),
   foreign key (pricecat) references pricecats (pricecat) );
```

That is, consider that the `titles` table contained an additional column `preview_days`. A value for `rental_days` is already stored in the `pricecats` table but the video store wishes 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 will use the `preview_days` value to override the default number of rental days. When the store manager is entering a new title, the application will give the manager an opportunity to supply a new value for the `preview_days` but will fetch the current `pricecats` value as the suggested value. To support this, the application must modify the Relations property for the link named in the validation link from:

```
pricecat      join      pricecat
```

to

```
pricecat      join      pricecat
preview_days  lookup    rental_days
```

Now transaction manager will generate the following select statements 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 `@null`, now the transaction manager selects the price category's rental days to the `preview_days` field 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 may choose to use the suggested value or may change the value in the `preview_days` field to any other value. When the title information is saved, the `preview_days` value is saved in the `titles` table. The `pricecats` table remains unchanged.

The transaction manager will use the lookup relation only when performing a validation link. It is ignored when the transaction manager executes a `VIEW` or `SELECT` command.

Transaction Models

Even though the transaction commands available in the transaction manager are the same for every database engine, the processing that occurs for the transaction commands varies from engine to engine. For this reason, the transaction manager puts the main processing for the transaction commands in the transaction model.

The transaction model is a program in C or JPL. JYACC distributes sample transaction models for each database engine it supports. There is at least one transaction model for each database engine; however, you can have different models for the same engine depending on the processing needed by your application.

Included in the name is an abbreviation which identifies the database engine. Some of the transaction models include `tmjdb1.c` for JDB, `tmora1.c` for ORACLE, `tmsyb1.c` for SYBASE, and `tminf1.c` for INFORMIX. These models are referred to as the standard transaction models.

The transaction model performs the detail work for the transaction manager. The transaction model can handle things that are application, engine or strategy specific. This includes the following:

- Cursor management strategies.
- Locking strategies.
- Data fetching strategies.
- Database transaction strategies.

You can modify the transaction model that is distributed with JAM; however, the transaction model is subject to change with each new release. You can also change the model behavior by writing your own transaction hook functions, a much simpler task than editing the model itself. If you do decide to modify the transaction model, you should be familiar with how the database engine processes database transactions, locking procedures and database cursors. For more information on writing transaction hook functions, refer to Chapter 22.

By default, the transaction manager accesses the standard transaction model developed for the specified engine. However, you can also set the transaction model in the screen editor as part of the table view properties or the screen properties.

Cursor Usage

A database cursor is a database object associated with a specific query or operation. Since cursors can be expensive in terms of memory, the way database cursors are used varies from engine to engine.

A JAM cursor may or may not correspond to a database cursor. With SYBASE, declaring a new JAM cursor opens a SYBASE `dbproc`—which is an expensive operation. Since declaring a JAM cursor is expensive in SYBASE, the transaction model does not close the JAM cursor at the end of each request. It reuses the cursor once it is declared and closes the JAM cursor when it closes the connection. Transaction models for other database engines close the JAM cursor after executing a statement.

Database Transaction Strategies

A database transaction is a series of steps that must be completed as a group. Once all of the steps are completed, the transaction is committed (i.e., saved) to the database. If all of the steps in a database transaction cannot be completed, the steps are rolled back so that none of the steps are completed. This keeps the database in a consistent state.

For example, in the `videobiz` database, checking out a video to a customer is a database transaction. To complete this transaction, you must be able complete three

steps: enter the rental of the video in the `rentals` table, update the `tapes` table to reflect the status of the video tape, and update the number of rentals in the `customers` table.

As a general rule, database engines use one of two database transaction strategies. One strategy uses `BEGIN`, `COMMIT`, and `ROLLBACK` commands to define a transaction. The second strategy just uses the commands `COMMIT` and `ROLLBACK` to define a transaction. Usually, engines using the second strategy also have access to an `AUTOCOMMIT` mode which commits every statement as it is executed.

Since database engines have different methods of implementing database transactions, the transaction model for a particular engine reflects those differences. Before editing the transaction model, you need to be familiar with the behavior of the database engine you are using.

The volume of database transactions in your application might also affect the processing contained in your transaction model. If you need to issue a `COMMIT` command frequently, you might need to change the transaction model to do that processing for you.

Since the transaction model is different for each engine, refer to the comments in the model's source code for a more complete description of the transaction model behavior.

Before Image Processing

When you execute a transaction manager command that could modify data, such as `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE`, the transaction manager initializes before image processing for all updatable table views. When before image processing is active, the transaction manager keeps track of any changes to the data displayed on the screen while keeping the previous value, the before image, in memory. Then, when you execute the `SAVE` command, the transaction manager translates any changes to the data into the SQL statements needed to update the database. Before image is initialized using the function `sm_bi_initialize`.

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 see what kind of change was made to the row or occurrence. The values of TM_OCC_TYPE are:

BI_DELETED	The 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.

If you choose not to save your changes to the database, the transaction manager throws away the changes, but the previous value is not restored to the screen.

You can query for the current value of TM_OCC_TYPE using the library function sm_tm_inquire. The standard transaction models use the library function sm_bi_compare to query for the type of change made to a row or occurrence. The values for TM_OCC_TYPE are defined in tmusubs.h.

Hook Functions

Transaction manager hook functions are available for changing the processing associated with any transaction event. A transaction manager hook function is passed one argument, the event name.

Hook functions are installed by entering the hook function name in the table view's Function property. You can have a hook function for each table view; however, for database queries, the hook function must be specified in the first table view of a server view.

The following return codes are available for transaction manager hook functions:

Return Value	Meaning
TM_OK	The event processing succeeded.
TM_FAILURE	The event processing failed.
TM_PROCEED	After completing the hook function, proceed as if this function had never been called. In the case of a transaction hook function, this means that the transaction model will be called.
TM_CHECK	Test to see if an error occurred. This is used in database-based transaction models to check for SQL execution errors.
TM_CHECK_ONE_ROW	In to an error test, test to see that exactly one row was affected.
TM_CHECK_SOME_ROWS	In addition to an error test, test to see that one or more rows were affected by the processing.
TM_UNSUPPORTED	The event was not recognized.

For more information about writing transaction hook functions, refer to page 384.

Customizing Transaction Manager

The transaction manager allows you to quickly prototype application screens. However, once the screens are built, you probably will want to modify the transaction manager processing to better suit the needs of your application.

Some of these customizations are relatively simple. Since instructions are included in other sections of the documentation, they are listed here only for reference.

- Adding your own menu bar.
- Specifying in a transaction manager command which portion of the table view tree receives the command.
- Editing the table view's Sort Widgets property to order the data fetched from the database.
- Specifying with the Updatable property whether the information from a database table can be changed.
- Editing the widget properties that control the SQL generation. Possible changes include displaying values of aggregate functions on the screen and differentiating which widgets participate in the various SQL statements and `WHERE` clauses.
- Adding fields containing computed values to a table view.

- Adding new styles and classes.

Some of the customizations are more difficult and are covered in the following sections:

- Controlling database locking to allow for concurrent users.
- Controlling error message display.
- Reusing and redeclaring cursors in the transaction manager.
- Controlling the display and update of database information.
- Querying and setting transaction manager values using the property API.
- Writing hook functions to change SQL generation, control command processing, and set widget properties at runtime.

Controlling Database Locking

Once information has been fetched from the database and it is time to update that information, you need to make sure that the data has not been updated by another user.

There are two basic strategies for dealing with this issue. One strategy uses the database engine locking facilities to prevent other users from fetching and/or modifying data. The other strategy, called optimistic locking, involves database design and the creation of a version field in the database tables. The version field can either be a specific data type provided by the engine, such as a timestamp, or it can be a numeric data type that is incremented with every update.

Implementing Optimistic Locking

Optimistic locking is based on the assumption that users rarely need to edit the same piece of data at the same time. Therefore, two or more users may access the same information. Checks are performed when the database is updated, not when the data is fetched.

In JAM, there are two basic methods for implementing optimistic locking. You can designate a widget to be a version column and let JAM add that column to the necessary statements, or you can designate a column to be added to the `WHERE` clause in a SQL `UPDATE` or `DELETE` statement.

In the first method, the version column in the database table corresponds to a widget having the Version Column property set to Yes and the C Type property set

to Int, Long Int, or Float. The value chosen for C Type property should match the column's data type. Once this occurs, during a SQL `INSERT` statement, the version column is included both in the column list and the `VALUES` clause. The value is automatically set to 1. For SQL `UPDATE` statements, the version column is automatically incremented by 1 to a new value in the `SET` clause while the previous value is included in the `WHERE` clause. For SQL `DELETE` statements, the version column is automatically included in the `WHERE` clause. For both `UPDATE` and `DELETE` statements, if the value in the `WHERE` clause matches the database value, the database modification is performed.

Note that for the widget having the Version Column property set to Yes, the In Delete Where and In Update Where properties must be set to No, and the Insert Expression and Update Expression must be blank.

The other method uses a data type provided by the database, such as a timestamp. For this method, set the In Delete Where and In Update Where properties to Yes for the widget corresponding to the column. The value fetched to the widget is added to the `WHERE` clause. If the value in the `WHERE` clause matches the database value, the database modification is performed.

You should not set the In Delete Where or In Update Where properties to Yes on a version column field. If Version Column is also set to Yes, the transaction manager will report the error `Version Column setting is incompatible with the properties In Delete Where, In Update Where`.

For examples using the Version Column, In Delete Where, and In Update Where properties, refer to Chapter 18 in the *Application Development Guide*.

Implementing Engine Locking

The standard transaction models distributed with JAM do not implement any pessimistic locking features; however, the `PRE_SELECT` and `POST_SAVE` events could be edited to implement engine locking using `dm_dbms` to pass the necessary commands to the database engine.

Error Processing 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:

- Setting the value of TM_STATUS.
- Error processing for transaction events.
- Setting error messages and error numbers in the transaction manager.

For information about the cause of transaction manager errors, refer to Chapter 24. For information about how hook functions deal with errors, refer to page 384.

TM_STATUS Variable

Testing the Value of TM_STATUS

You can test for the current value of TM_STATUS using the library function sm_tm_inquire(TM_STATUS).

The default processing by the transaction manager will set 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 model or hook function may set the return value unconditionally by calling:

```
sm_tm_iset(TM_STATUS, return_value)
```

However, the standard 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 standard models also use this after they call the function dm_dbms to execute SQL statements. This preserves non-zero values that may 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

Processing for Requests

Most of the transaction manager commands are subdivided into three transaction requests:

- TM_PRE_*command-name*
- TM_*command-name*
- TM_POST_*command-name*

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 PRE_ and POST_ requests, but will prevent processing of the "main" request. Even if an error occurs in PRE_ or 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 PRE_ processing. Traversal for POST_ processing begins immediately after "main" processing ceases.

Let's look at the processing of the three requests associated with the VIEW command: TM_PRE_VIEW, TM_VIEW, and TM_POST_VIEW. 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 for the Event Stack

When an error occurs (setting TM_STATUS to a non-zero value), any transaction events on the event stack continue to be processed. Since all the 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. If desired, 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.

Even though this is the standard behavior for errors in the transaction manager, it can be changed by setting the following variables:

- TM_EMSG_USED — Controls whether the error message is displayed.

- TM_MSG_TEXT — Unconditionally sets the error message text.
- TM_PROPOSE_MSG_TEXT — Sets the 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

You can set whether you want the transaction manager error message to be displayed using:

```
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 was 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 exit, the error handler displays the database driver error, but disables the duplicate transaction manager error.

```
proc entry
DBMS ONERROR JPL dberror
return

proc dberror
if @dmretcode == DM_NO_CONNECT
{
   call sm_tm_iset(TM_EMSG_USED, 1)
   msg emsg ”JAM DB: ” @dmretcode ” ” @dmretmsg ”%N”\
   ”Engine Error: ” @dmengretcode ” ” @dmengretmsg}
else
   msg emsg ”JAM 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 gets displayed when sm_tm_command finishes.

The standard models also use this call after they call the function dm_dbms to execute SQL statements. They thus preserve non-empty values that may have been set by a 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 is a message number on the message file. TM_MSG is used to set 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 transaction manager error messages are located in the JAM message file. The message numbers occur in two groups, one ranging in value from 53308 to 53013 and the other ranging in value from 53376 to 53423.

Suppressing Transaction Manager Error Messages

If you want to suppress the transaction manager messages, you can call the sm_tm_iset(TM_EMSG_USED, flag) function in the standard transaction model, in the database error handler, or in a hook function. If flag is set to a non–zero value, the transaction manager does not display an error message.

Controlling Cursor Behavior

The transaction events generally declare and close any JAM cursors needed to perform command processing. However, in some cases, special treatment of cursors may be needed.

Declaring Cursors

Generally, a DECLARE CURSOR statement is associated with a cursor name and a SQL statement. However, in the transaction manager, other formats are available in order to modify the processing for SELECT statements.

Instead of specifying the cursor name, you can use the variable @tm_sel_cursor. When this variable is colon expanded, the name of the default SELECT cursor used by the transaction manager is supplied. For example:

```
DBMS DECLARE :@tm_sel_cursor CURSOR FOR SELECT ...
```

Also, the DECLARE CURSOR statement can be issued without a SQL statement. If you use this format, the DECLARE CURSOR statement merely adds the cursor to the list of open cursors. Later, the cursor can be redeclared with the necessary SQL statement. This allows you to take advantage of commands like DBMS CATQUERY which maps a series of select expressions into one JAM target. The following example uses this command:

```
proc catquery_hook (event)

if (event == TM_SEL_BUILD_PERFORM)
{
    DBMS DECLARE :@tm_sel_cursor CURSOR
    DBMS WITH CURSOR :@tm_sel_cursor CATQUERY
    return TM_PROCEED
}
...

return TM_PROCEED
```

Using the Default Cursors

When you connect to a database engine, JAM's database drivers automatically create default cursors to use for SQL statements. Generally, one of the cursors is used for database queries and performs the SQL SELECT statements. The other is used for database modifications and performs any SQL UPDATE, INSERT and DELETE statements.

Depending on the engine, creating a JAM cursor can be very expensive. In SYBASE, for example, each JAM cursor is a dbproc. For this reason, the standard transaction model for SYBASE takes the default cursor initiated in the database driver for transaction manager processing, for its exclusive use. Also, the transaction manager for SYBASE does not automatically close all the cursors until a FINISH command is executed.

The flags controlling the use of cursors are defined in the transaction model. If you wish to edit these flags, modify the model and recompile the executable.

Displaying Data

The transaction manager automatically generates the necessary SQL statements for displaying and updating data based on the properties for each widget on the screen. For a description of the widget properties that can be used to control SQL generation, refer to Chapter 18.

Selecting Data

Two commands select data from the database: `VIEW` and `SELECT`. With the `VIEW` command, the data is for display only; no updates are allowed. With the `SELECT` command, updates are allowed.

Scrolling through the Select Set

The `CONTINUE` command 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. Usually you will create scrolling JAM widgets or a JAM grid for displaying the data. In environments where memory is limited, you may fetch only a small number of rows to the JAM may fetch only a small number of rows to the JAM application and buffer the rest in a file on disk. 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_BACK` command displays the previous set of data, and issuing a `CONTINUE_TOP` command displays the first set of data.

Note that JAM does *not* set backward scrolling via continuation files as the default since JAM 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.

Setting the Fetch Directions Property

The Fetch Directions property has four possible values:

- `Down Only–all modes`
- `Up/Down–view mode`
- `Up/Down–all modes`
- `–default–`

If the value is `Down Only–all modes`, only the `CONTINUE` command is valid. For the `CONTINUE` command, the transaction manager issues a `TM_CONTINUE` request.

If the value is `Up/Down–view mode`, the following commands are available in `view` mode in addition to `CONTINUE`:

- `CONTINUE_BOTTOM`
- `CONTINUE_DOWN`
- `CONTINUE_TOP`
- `CONTINUE_UP`

If the value is `Up/Down–all modes`, these commands are available in `view` and `update` modes. Remember that if you set Fetch Directions to this value, you must re-fetch the data after a `SAVE` command, so that the user can view the updates and so that before-image processing can be re-initialized. JAM does not update the continuation file. If a `SAVE` command is not executed and the user redisplays the data with any of the `CONTINUE_` commands, the original, un-modified data is displayed, and the transaction manager has no mechanism to prevent the user from updating the row a second time.

If the table view's Fetch Directions property is specified as `default`, the screen's Fetch Directions property is consulted to see if the commands are valid. If the screen's Fetch Directions property is also specified as `default`, this is the equivalent of `Down only–all modes`.

Finding the Number of Rows Fetched

There are actually two methods for finding out the number of rows fetched. The first method calls `sm_tm_inquire(TM_OCC_COUNT)`. This gives you the number of rows fetched for the current table view. You can test for this in the `TM_SEL_CHECK` event.

An alternate method is to write a hook function to test for the value of `@dmrow-count`. Since the transaction manager uses `DBMS` commands in the database driver to fetch and update data, you can test for the value of any of the global variables in the database driver. However, since the `@dm` variables are cleared before each `DBMS` command, you must copy this value to another location if you need it for further processing.

For the transaction manager commands `VIEW` and `SELECT`, you can test for number of rows fetched by writing a hook function for the `TM_SEL_CHECK` event. Remember to set the hook function on the server view, with a server view being the first table view in a group of table views joined by server links. Otherwise, the hook function is not called.

```
proc num_rows (event)
if event == TM_SEL_CHECK
    {
    vars retcode
    retcode = @dmrowcount
    }
return TM_PROCEED
```

The check for the value of `@dmrowcount` is performed in the `TM_SEL_CHECK` event since this event occurs after the SQL statement has been generated in `TM_SEL_BUILD_PERFORM`, but before the next `DBMS` command which resets the variable.

Selecting Data for Update

Since the `SELECT` command fetches data for possible database updates, the primary key fields for any updatable table views must be on the screen. If a primary key column is not a member of the current table view, it must be a member of one of the parent table views.

Also, all the widgets in a server view having the Use In Update property set to Yes must have the same number of onscreen occurrences and the same number of maximum occurrences. If not, the error message will say that the table view is not synchronized.

Note that for primary key fields, if you edit the primary key value, the standard transaction models issue a SQL `DELETE` statement for the old value followed by a SQL `INSERT` statement for the new value.

Synchronizing the Table View

For the `SELECT` command, each member of the table view must have the same number of occurrences in order for automatic synchronization of the table view to occur. When a table view is synchronized, you can easily insert and delete data in each occurrence. For the `VIEW` command, members no not need the same number of occurrences since the data is not updated.

To override this behavior for the `SELECT` command, you can set the Synchronization property to No for any widget. This allows the SQL `SELECT` statement to be executed. However, setting this property to No does not change how JAM fetches data. For any SQL `SELECT` statement, the number of rows fetched equals the least number of occurrences for any widget in the server view. If one widget in the table view has three occurrences and another widget in the same table view has five occurrences, JAM fetches three rows.

Remember that a Multiline Text widget with Word Wrap set to Yes is considered to be a single occurrence so that only one row is fetched for the server view.

Deleting Data

After a `CLEAR` command clears data from the screen, the `SAVE` command in the standard transaction models generates SQL `DELETE` statements in order to delete

those rows from the database. If the widgets affected by the `CLEAR` command are arrays or grids, a statement is issued for each onscreen occurrence. To clear a single occurrence in an array or grid, press the DELL logical key or call the `sm_doccur` function to delete the line of data. Of course, the table view must be updatable in order for the statements to be generated.

Note that the standard transaction models will issue a SQL `DELETE` statement for the row any time the primary key widgets have been cleared. This allows you, in addition to the above methods, to use the property API to find the primary key fields so that you can programmatically clear them in order to delete the data.

Using Traversal Properties

In JAM, it is the property API that is used to set property values at runtime. When you are using the transaction manager, you can use the property API syntax to query for information about objects in the current traversal tree, like server views. These properties often have no equivalent in the screen editor's Properties window. Since they contain information about the traversal tree, they are called *traversal properties.*

Note that a table view is defined to be a server view if:

- It is the root table view.
- A sequential link connects this table view to its parent, not a server link.

If you use a table view instead of a server view with server view properties, no error is reported. Instead, it returns the information for just that table view. For example, if you used a table view for `num_svs_below` (number of server views at or below this server view), the value returned is 1 which is derived from the table view itself.

The following categories and types of information are available:

- Fields
  - `tv` — The table view containing the specified field.
  - `sv` — The server view containing the specified field.
- Table views
  - `sv` — The server view containing the specified table view.
  - `num_fields` –The number of fields in the specified table view.

- `field` — The field in the table view corresponding to the specified table view and field number. Note that the table view field number specified here does not correspond to the JAM field number.
- `parent` — The parent table view for the specified table view. Since the property does not apply for the root table view, JAM returns an empty string if the root table view is specified.
- `parent_link` —The link name where the specified table view is the child. Since the property does not apply for the root table view, JAM returns an empty string if the root table view is specified.
- `num_children` —The number of child table views for the specified table view.
- `child` — The child table view corresponding to the specified table view and number.
- `num_key_columns` — The number of primary key columns for the specified table view. For the names of the primary keys, use the `primary_keys` property.

  Note that this information is only available for updatable table views after a `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE` command. Otherwise, you get an error. If this is specified in a JPL procedure, you receive the `Bad field name, #, or subscript` message.

  The primary key columns may not be members of the table view itself. They can appear in a parent or grandparent table view.
- `key_field` — The primary key field for the specified table view and primary key number. This is only available for updatable table views after a `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE` command.
- `key_constant` — The primary key constant for the specified table view and constant number. This is only available for updatable table views after a `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE` command.
- `num_sorts` — The number of entries in the Sort Widgets property of the table view.

◦ Server views

- `bi_status` — The occurrence number of the server view. This property is used in conjunction with the following flags: `PV_BI_FETCHED`, `PV_BI_EMPTY`, and `PV_BI_CHANGED`.

  `PV_BI_FETCHED` indicates that the before image of the row was fetched from the database. `PV_BI_EMPTY` indicates that the before image of the

row is empty; therefore, the occurrence was inserted by the user. `PV_BI_CHANGED` indicates that the before image of the row has been edited.

- `num_svs_below` — The number of server views at and below the specified server view. If a table view which is not a server view is specified, the integer returned is for the specified table view and its direct children.
- `field_below` — The field name in or below the specified server view which corresponds to the specified field number.
- `num_fields_below` — The number of the fields in and below the specified server view.
- `num_sv_fields` — The number of fields in a server view.
- `num_tvs` — The number of table views in a server view. If the table view is not the first table view in the server view, the integer returned is for the specified table view and its direct server view children.
- `num_tvs_below` — The number of table views at and below the specified table view.
- `source_occ` — The occurrence in the parent table view that was valid when the child table view was last fetched.
- `sv_field` — The field name for the specified server view and field number.
- `sv_below` — The server view at or below the specified server view which corresponds to the specified server view number.
- `tv` — The table view in a server view which corresponds to the specified server view and table view number.
- `tv_below` — The table view at or below the specified table view which corresponds to the specified table view number.

○ Links

- `num_relations` — The number of entries in the Relations property of the link.

Note that 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 `Bad field name, #, or subscript` message.

Finding Table View and Server Views

It is important to understand how the property API processes table views and server views. Let's look at a DB Interactions screen which graphically illustrates table views and server views.

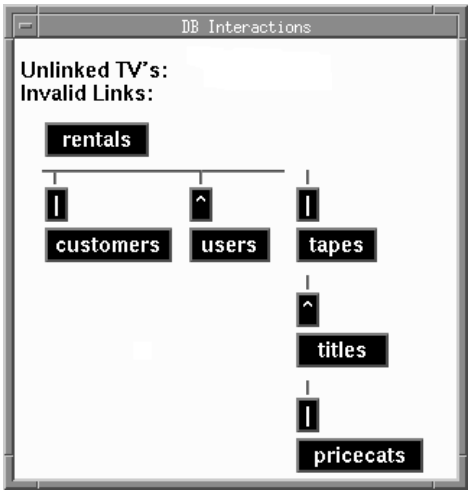


Figure 29. DB Interactions screen illustrating a series of table views and server views.

In this screen, there are three server views: `rentals`, `users`, and `titles`—`rentals` because it is the root table view, `users` and `titles` because they have sequential links to their parent table view.

The property `num_svs_below` is the number of server views at or below the specified server view. If `rentals` is specified, the property API returns 3 servers views: `rentals`, `users` and `titles`. However, if `customers` is specified, the property API returns 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 `customers` itself.

Examples

The example is a simple one. On field entry, it finds which table view the current field is a member of and executes the transaction manager `VIEW` command specifying that table view.

```
proc get_tv_query
if K_ENTRY
{
   vars value1
   value1 = name->tv
   call sm_tm_command("VIEW :value1")
}
return
```

Writing Hook Functions

A transaction hook function replaces part of the functionality provided by a transaction model. You can write transaction hook 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 hook function, you need to:

- Write a hook function which includes:
  - The event whose functionality you want to modify or replace.
  - The processing to be added to the event.
  - The return code which tells the transaction manager how to proceed.
- Install the hook function by setting the table view's Function property to the name of the JPL procedure containing the function.

This section explains how to write a simple hook function, how to specify the appropriate return code, how to modify processing for SQL `SELECT` statements, and how to modify processing for SQL `INSERTS`, `UPDATES` and `DELETES`.

Writing a Simple Hook Function

One use of hook functions is to provide hand-coded SQL to replace generated SQL. Here is a simple hook function that executes a SQL statement:

```
proc my_simple_hook (event)
if event == TM_VIEW || event == TM_SELECT
{
   DBMS SQL \
      SELECT title_id, name, genre_code
   return TM_CHECK
}
return TM_PROCEED
```

For this example, the Function property on the `titles` table view is set to `my_simple_hook`. The hook function is called each time an event is processed for this table view.

Since transaction hook functions are passed one argument, the transaction event, you need to specify which events you want to modify. In the example, 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 hook function had not been called, so that the transaction model's processing is performed.

With the sample hook function, for `TM_VIEW` and `TM_SELECT`, the transaction manager uses the database driver's `DBMS_SQL` command to retrieve data from `titles` table. The return code `TM_CHECK`, which tells the transaction manager to test for an error, is used so that the transaction manager can check for database errors from the `SELECT` statement.

Specifying a Return Code

It is important to specify the correct return code at the end of your hook function since it tells the transaction manager what to do next. The following table summarizes the possible return values.

Return Value	Meaning
`TM_FAILURE`	The event processing failed.
`TM_OK`	The event processing succeeded.
`TM_PROCEED`	Proceed to the next step. For hook functions, this is to call the transaction model.
`TM_CHECK`	A test should be made to determine if an error occurred.
`TM_CHECK_ONE_ROW`	A test should be made to determine if an error occurred. Furthermore, the event processing succeeded only if exactly one row was affected.
`TM_CHECK_SOME_ROWS`	The transaction model should be consulted to determine if an error occurred. Furthermore, the event processing succeeded only if one row or more rows were affected.
`TM_UNSUPPORTED`	The event was not recognized.

Specifying TM_PROCEED

Let's look at what happens if the SQL statement has a return code of TM_PROCEED:

```
if event == TM_VIEW || event == TM_SELECT
{
   DBMS SQL \
      SELECT title_id, name, genre_code FROM titles
   return TM_PROCEED
}
```

Since TM_PROCEED tells the transaction manager to go ahead and call the transaction model, as if the hook function had not been called, the transaction model's processing would *also* be performed after the DBMS SQL statement was executed. The processing for these requests in the standard 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 property is set to Yes, generates the SQL for the SELECT statement, and then executes the statement.

Since the processing for all the events takes place quickly, you might not visually notice that both sets of processing are performed unless you view the processing in the debugger and break on each transaction manager event.

Note that if both a hook function and a transaction model is called for a single event by using TM_PROCEED, the processing in the hook function takes place *before* the processing in the model.

Specifying TM_OK

The return code in the sample statement could be set to TM_OK. TM_OK is used when the processing is contained in itself, needs no processing performed by the model, and needs no checking for database errors. Since the sample statement does not need any processing in the model, this return code is one possibility.

```
if event == TM_VIEW || event == TM_SELECT
{
   DBMS SQL \
      SELECT title_id, name, genre_code FROM titles
   return TM_OK
}
```

Even though the return code TM_OK does not check for database errors, the standard transaction models set TM_STATUS to –1 whenever the database error handler returns an error. If the sample statement caused the database error handler to be invoked, for example by specifying an invalid column name, first the database error handler returns an error. Then, the transaction manager returns an error that a transaction model or hook function had an error.

```
if event == TM_VIEW || event == TM_SELECT
{
   DBMS SQL \
      SELECT title_id, title_name, genre_code FROM titles
   return TM_OK
}
```

Checking for Database Errors

Three return codes are designed specifically for checking for errors from the database engine and from JAM's database drivers: TM_CHECK, TM_CHECK_ONE_ROW, and TM_CHECK_SOME_ROWS. TM_CHECK is used to check for any error from JAM'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 stack for this return code. This is the best return code for the sample statement.

```
if event == TM_VIEW || event == TM_SELECT
{
    DBMS SQL \
        SELECT title_id, name, genre_code \
        FROM titles WHERE title_id = :+title_id
    return TM_CHECK
```

TM_CHECK_ONE_ROW pushes the TM_TEST_ONE_ROW event onto the stack. This event checks the value of @dmrowcount, a global variable available in JAM's database drivers, 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 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 about transaction manager error processing, refer to page 371.

Performing Error Checking

In the previous section, the sample hook function, my_better_hook, replaced a SQL SELECT statement. A slight variant is presented below. The difference is that the variant does its own checking for errors. In this case, 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 my_checking_hook( 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
    return TM_OK
}
return TM_PROCEED
```

Unsupported Events

The standard 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

An earlier section described a simple hook function that replaced the automatically generated SQL with a SQL `SELECT` statement. You could use that format when replacing the entire `TM_VIEW` or `TM_SELECT` request; however, many times it would be easier to just modify the generated SQL, not replace it. In order to do this, you need to understand the commands, requests and slices that work together to display data in the transaction manager.

Two transaction manager commands fetch data from the database: `SELECT`, which allows you to update the selected data, and `VIEW`, which displays data for viewing purposes only. Three requests that are generated for the `VIEW` command—`TM_PRE_VIEW`, `TM_VIEW`, and `TM_POST_VIEW`. Similarly, three requests are generated for the `SELECT` command—`TM_PRE_SELECT`, `TM_SELECT`, and `TM_POST_SELECT`. In the standard models, even though all three requests are generated for each command, the processing is performed in the main request for each command. For the `SELECT` command, the main request is `TM_SELECT`, and for `VIEW`, it is `TM_VIEW`.

The events are defined to encompass a small enough chunk of processing to make replacement of a single event a fairly simple task. However, there is much more processing associated with the `TM_VIEW` and `TM_SELECT` requests than is suggested by `my_simple_hook` function in the previous section. The additional processing is related to the fact that the model is designed for use with a database, whereas the transaction manager is built with more general purposes in mind (e.g. use with on-line transaction processing systems). To accommodate the needs of specific transaction models, each model is allowed to generate additional events while handling an event. A model's ability to generate events is important, because it allows the model designer to package functionality into small, re-usable chunks. The standard models generate the following events in response to receipt of `TM_VIEW` and `TM_SELECT` events. In a manner of speaking, the transaction model "slices" the transaction manager request into "smaller" events. These events are referred to as *slices.*

The transaction manager always generates one type of request per table view traversal, and typically generates only a single request. For example, all table views will receive a `TM_PRE_VIEW` event before any table view receives a `TM_VIEW` event (unless errors are encountered). On the other hand, the slices generated in response to a single request are always received by a table view before the traversal process continues.

The following slices are generated for the TM_VIEW and TM_SELECT events:

- TM_GET_SEL_CURSOR — Allocate a JAM cursor for use by the SELECT statement. Depending on the database, a JAM 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 — Generate data structures that will be 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 — Build and execute the SQL statements needed to view the data. Use the JAM cursor allocated in the TM_GET_SEL_CURSOR step.
- TM_SEL_CHECK — Check to determine whether to give up the JAM 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

Here is a another hook function that executes a SQL statement. It is better than the my_simple_hook example because it uses the transaction model's cursor management and error reporting capabilities. The special variable @tm_sel_cursor contains the name of the cursor to be used to execute the SQL SELECT statement.

The hook function is called for the TM_SEL_BUILD_PERFORM event. This event was chosen since it builds and executes the SQL statements. Since this slice is generated for both the TM_SELECT and TM_VIEW requests, this hook function will be called for either request.

```
proc my_better_hook (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
   return TM_CHECK
}
return TM_PROCEED
```

In some cases, you may want to check for errors by writing error checking code within the hook function. In that case, you should return TM_FAILURE if an error is encountered, and TM_OK if there is no error.

Note that `my_better_hook` permits the `TM_SEL_GEN` slice to be handled by the transaction model, which (in the case of the standard models) will have JAM build unneeded data structures. For slightly better performance, that slice can be skipped as follows:

```
proc my_faster_hook (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
   return TM_CHECK
}
if event == TM_SEL_GEN
   return TM_OK
return TM_PROCEED
```

Modifying SQL Generation

In addition to writing hook functions which replace the SQL `SELECT` statement, you can also write hook functions to modify the automatic SQL generation. Use one of the following C functions which 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.

For more information on each function, refer to the *Language Reference*. A sample hook function which adds a column and its corresponding table to the `SELECT` statement is shown below.

```
proc titles_hook (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 has been modified or new data has been entered. In total, the transaction manager may 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 standard models do nothing.
- TM_DELETE — Standard models generate SQL DELETE statements for records to be deleted; one per modified record.
- TM_UPDATE — Standard 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 — Standard models generate SQL INSERT statements for records to be updated; one per entered record.
- TM_POST_SAVE — Standard 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.

The reason that TM_DELETE is generated before TM_UPDATE and TM_INSERT is to prevent duplicate record errors. This is also why TM_UPDATE is generated before

TM_INSERT. Note also that a single cursor is used for all save operations. This permits all save operations to be part of the same database transaction.

The standard models further slice the TM_DELETE, TM_UPDATE, and TM_INSERT requests. Note that slicing is performed only if the slices are needed. For example, when a row is being inserted, no slices are generated for the TM_DELETE request. The three slices for each of these requests are:

- TM_GET_SAVE_CURSOR — This event is 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 — Generate the SQL statement and use 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 — Execute the declared cursor.

Supplying custom INSERT, UPDATE, and DELETE statements should normally be done in the TM_*request*_DECLARE events, since they will occur only when a new cursor must be declared (which can be a somewhat expensive operation, depending on the database).

Here's a hook function that provides custom SQL INSERT, UPDATE, and DELETE statements:

```
proc my_save_hook( 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 standard models execute the cursor whose name is stored in `@tm_save_cursor`.

In the execution of the cursor, the bind variables (e.g. `::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 Chapter 18.

Transaction Manager Commands

This chapter describes the `sm_tm_command` function and the transaction commands that can be called using this function. The section for each command contains the following information:

- Syntax — Lists the command and its parameters.
- Description — Gives an explanation of the command.
- Sequence — Lists other transaction manager commands that may be needed before or after this command.
- Requests — Lists the transaction requests and slices that can be generated with a command. This information is useful when writing a transaction hook function to change the processing in a request or when modifying the transaction model. For information on writing transaction hook functions, refer to page 384.

Some requests refer to the following transaction attributes:

`TM_FULL`	Indicator of whether it is a full (1) or partial (0) command.
`TM_OCC`	Occurrence number being processed.
`TM_OCC_COUNT`	The number of occurrences in the table view.
`TM_STATUS`	Error indicator.
`TM_VALUE`	General purpose integer.

Use the library functions `sm_tm_inquire`, `sm_tm_pinquire`, or `sm_tm_pcopy` to test the value of transaction attributes. Use the library functions `sm_tm_iset` and `sm_tm_pset` to set transaction attributes. All of these library functions are described in Chapter 6 of the *Language Reference*.

sm_tm_command

Executes a transaction command

```
#include <tmusubs.h>

int sm_tm_command (cmd_string);
```

`cmd_string` Contains one of the following transaction commands and its associated parameters:

CHANGE	CONTINUE_DOWN	COPY_FOR_VIEW	REFRESH
CLEAR	CONTINUE_TOP	FETCH	SAVE
CLOSE	CONTINUE_UP	FINISH	SELECT
CONTINUE	COPY	FORCE_CLOSE	START
CONTINUE_BOTTOM	COPY_FOR_UPDATE	NEW	VIEW

The parameters may include a table view name and/or command scope. Refer to the following sections in this chapter for each command's syntax.

Returns

- `STATUS` of the current transaction.

Description

`sm_tm_command` executes the specified transaction manager command.

When specifying a command, the table view name is case sensitive; however, the command name and the optional parameters following the table view name are not case sensitive.

By definition, a command is in progress from the moment `sm_tm_command` is called until the moment it returns. As it processes most commands, `sm_tm_command` invokes transaction hook functions and transaction models. These, in turn, should not invoke transaction manager commands, because the transaction manager cannot process its commands recursively. This implies that they should not close the active screen (which triggers a `FINISH` command), or cause any other screen to be displayed that contains table views (which triggers a `CHANGE` command).

Transaction Modes After recognizing a transaction command, the transaction manager either sets the transaction mode or checks the transaction mode to see if the specified command is available with the current mode. If the command is not supported in the current

mode, or if the command is not recognized, then the transaction manager displays an error message that the mode does not permit the specified command. It also sets the value of TM_STATUS to –1, which causes sm_tm_command to return a value of –1. For more information on command availability in transaction modes, refer to page 346.

Tree Traversal

Most commands traverse the table views in a particular order to issue requests to transaction models and hook functions. The most common order is referred to as table/server view order. A server view is defined as:

- A single table view having no server links to other table views.
- A group of table views connected by server links.

Tree traversal in table/server view order begins at the root table view or at the specified table view. The traversal covers all table views within the server view, and then moves on to the next server view. The Parent and Child properties for each link help determine the traversal order. The tree traversal reaches a parent table view before its child, but there may be intervening table views (in the same and different server views).

Restriction

A transaction manager transaction must be in progress in order to call commands. Transactions are created with the START command which is called automatically on screen entry. However, the JAM events that occur on screen entry call the unnamed JPL procedure *before* calling the START command. For this reason, transaction manager commands cannot be invoked in the unnamed procedure.

Example

```
int sm_tm_command (”SELECT titles BELOW_TV”);
```

Errors

Errors in the transaction manager set TM_STATUS to –1.

In addition, there are return values for transaction models or transaction hook functions that set the value of TM_STATUS. The following table lists the return codes, the events that get generated for each return code, and the processing that occurs for the event.

Table 28. Return values for transaction hook functions and transaction models.

Return Code	Event	Processing
TM_OK	None	Do not invoke the transaction model.
TM_PROCEED	None	Invoke the transaction model.
TM_FAILURE	TM_NOTE_FAILURE	Call sm_tm_failure_message.
TM_UNSUPPORTED	TM_NOTE_UNSUPPORTED	Call sm_tm_failure_message.
TM_CHECK	TM_TEST_ERROR	Call sm_tm_dbi_checker.
TM_CHECK_ONE_ROW	TM_TEST_ONE_ROW	Call sm_tm_dbi_checker.
TM_CHECK_SOME_ROWS	TM_TEST_SOME_ROWS	Call sm_tm_dbi_checker.

Requests

Once you select a transaction command, the transaction manager generates the transaction events defined for that command. These events are defined to perform the processing needed for the command. The major events for each command are called requests. Some requests are further subdivided into more events, called slices, by transaction models or user hook functions.

The transaction manager has an event stack, onto which the transaction events are pushed. As the events are processed, they are popped from the stack. For more information on the event stack, refer to page 337.

The following table lists all the transaction manager events. A description of the general processing performed by each request or slice is part of the documentation for a command in which it is used. To see the processing done for a particular database engine, refer to the transaction model for that engine. For a summary list of the commands, requests, and slices, refer to page 340.

Table 29. List of transaction events available in the transaction manager. For each request or slice, the command where they are documented is also listed.

Event	Command
TM_CLEAR	CLEAR
TM_CLOSE	CLOSE
TM_CONTINUE_BOTTOM	CONTINUE_BOTTOM

Event	Command
TM_CONTINUE_DOWN	CONTINUE_DOWN
TM_CONTINUE_TOP	CONTINUE_TOP
TM_CONTINUE_UP	CONTINUE_UP
TM_COPY	COPY
TM_COPY_FOR_UPDATE	COPY_FOR_UPDATE
TM_COPY_FOR_VIEW	COPY_FOR_VIEW
TM_DELETE	SAVE
TM_DELETE_DECLARE	SAVE
TM_DELETE_EXEC	SAVE
TM_DISCARD	CLOSE
TM_FETCH	FETCH
TM_FINISH	FINISH
TM_GET_SAVE_CURSOR	SAVE
TM_GET_SEL_CURSOR	SELECT
TM_GIVE_UP_SAVE_CURSOR	SAVE
TM_INSERT	SAVE
TM_INSERT_DECLARE	SAVE
TM_INSERT_EXEC	SAVE
TM_NEW	NEW
TM_NOTE_FAILURE	Part of error processing
TM_NOTE_UNSUPPORTED	Part of error processing
TM_POST_CLEAR	CLEAR
TM_POST_CLOSE	CLOSE
TM_POST_COPY	COPY
TM_POST_COPY_FOR_UPDATE	COPY_FOR_UPDATE
TM_POST_COPY_FOR_VIEW	COPY_FOR_VIEW
TM_POST_NEW	NEW

Event	Command
TM_POST_SAVE	SAVE
TM_POST_SAVE1	SAVE
TM_POST_SAVE2	SAVE
TM_POST_SELECT	SELECT
TM_POST_VAL_LINK	Part of validation link processing
TM_POST_VIEW	VIEW
TM_PRE_CLEAR	CLEAR
TM_PRE_CLOSE	CLOSE
TM_PRE_COPY	COPY
TM_PRE_COPY_FOR_UPDATE	COPY_FOR_UPDATE
TM_PRE_COPY_FOR_VIEW	COPY_FOR_VIEW
TM_PRE_NEW	NEW
TM_PRE_SAVE	SAVE
TM_PRE_SELECT	SELECT
TM_PRE_VAL_LINK	Part of validation link processing
TM_PRE_VIEW	VIEW
TM_QUERY	CLOSE
TM_SAVE	SAVE
TM_SEL_BUILD_PERFORM	SELECT
TM_SEL_CHECK	FETCH
TM_SEL_GEN	SELECT
TM_SELECT	SELECT
TM_START	START
TM_TEST_ERROR	Part of error processing
TM_TEST_ONE_ROW	Part of error processing
TM_TEST_SOME_ROWS	Part of error processing
TM_UPDATE	SAVE

Event	Command
TM_UPDATE_DECLARE	SAVE
TM_UPDATE_EXEC	SAVE
TM_VAL_BUILD_PERFORM	Part of validation link processing
TM_VAL_CHECK	Part of validation link processing
TM_VAL_GEN	Part of validation link processing
TM_VAL_LINK	Part of validation link processing
TM_VIEW	VIEW

CHANGE

Switches to another transaction

```
int sm_tm_command ("CHANGE transaction-name");
```

transaction-name	The name of a valid transaction manager transaction.

Description

`CHANGE` switches to another transaction, making it the current transaction. To use this command, you must specify the transaction name. If the transaction does not exist, the previous transaction remains active. In cases where you move between two screens, the command is automatically issued as part of JAM's screen processing.

For the current transaction name, call `sm_tm_pinquire(TM_TRAN_NAME)`.

To specify a new transaction, use the `START` command.

Requests

There are no requests generated by the `CHANGE` command.

Example

For an example which also uses the `START` command, refer to page 455.

CLEAR

Clears data in widgets

```
int sm_tm_command ("CLEAR [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CLEAR` clears the data displayed on the screen for any widget belonging to a valid table view. `CLEAR` has two major uses:

- Clear onscreen data so that you can enter selection criteria for a subsequent `VIEW` or `SELECT`.
- Clear onscreen data so that `SAVE` processing will delete the database rows represented.

In order to delete rows from the database, the table view must be updatable. If the table view is non-updatable, the data is cleared from the screen, but SQL DELETE statements are not issued.

The CLEAR command does not change the transaction mode.

Push buttons and menu selections for the CLEAR command may choose to set the Class property to clear_button. By default, clear_button is active in all transaction modes.

Sequence

To delete rows, CLEAR must be followed by the SAVE command.

To perform a query-by-example, execute CLEAR before entering a value for the SELECT or VIEW commands.

Requests

The following requests can be generated by the CLEAR command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

Table 30. Main transaction manager requests for the CLEAR *command.*

Request	Traversal	Typical Processing
TM_PRE_CLEAR	By table/server view from the specified table view	Do nothing
TM_CLEAR	By table/server view from the specified table view	Do nothing (sm_tm_clear is called for the table view by the transaction manager after this request)
TM_POST_CLEAR	By table/server view from the specified table view	Do nothing

CLOSE

Closes the current database transaction, allowing you to discard or save your changes

```
int sm_tm_command ("CLOSE [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

If you have made changes in the table views on which `CLOSE` operates after a `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE` command, `CLOSE` displays a dialogue box which allows you to discard any changes entered. If you press `OK`, the changes are discarded. If you press `Cancel`, you will be returned to the screen so you can save your changes.

`CLOSE` sets the transaction mode to `initial` unless a table view is specified. In the default styles file, the style assigned to initial mode clears any protections on the widgets.

Push buttons and menu selections for the CLOSE command may choose to set the Class property to close_button. By default, close_button is inactive in initial mode but active in all other modes.

Sequence

This command is useful after SELECT, NEW, COPY, or COPY_FOR_UPDATE in order to discard your changes.

Requests

Table 31. Transaction manager requests for the CLOSE *command if changes have been made to the screen.*

Request	Traversal	Typical Processing
TM_PRE_CLOSE	By table/server view from the specified table view	Note that CLOSE or SAVE processing is beginning. (Processing identical for TM_PRE_SAVE)
TM_CLOSE	By table/server view from the specified table view. Traversal ends if TM_VALUE is set to TM_DISCARD_ACTION or TM_EXIT_ACTION.	Appropriate responses are those listed for TM_QUERY below, but typical processing is to do nothing.
TM_QUERY	By table/server view from the specified table view, but restricted to table views, if any, in which there has been a change that would entail a SAVE command. Traversal ends if TM_VALUE is set to TM_DISCARD_ACTION or TM_EXIT_ACTION	A message is chosen according to the value of TM_FULL. If 1, the displayed message is for the complete table view tree. If 0, the message is for a portion of the tree. sm_message_box, which displays the message, gives a choice of OK and CANCEL. TM_DISCARD_ACTION and TM_EXIT_ACTION are the corresponding values passed back to TM_VALUE.

Request	Traversal	Typical Processing
TM_DISCARD	By table/server view from the specified table view	Set a discard flag, consulted by TM_POST_SAVE1
TM_POST_CLOSE	By table/server view from the specified table view	Slices: TM_POST_CLOSE, TM_POST_SAVE1, TM_POST_SAVE2. These slices are described under the SAVE command, but no save cursor will exist.

The TM_CLOSE and TM_QUERY requests have four possible return values: TM_NO_ACTION, TM_DISCARD_ACTION, TM_SAVE_ACTION, and TM_EXIT_ACTION. The standard transaction models use two of these return values:

- TM_DISCARD_ACTION, which discards the changes to the data
- TM_EXIT_ACTION, which returns you to the screen in order to choose the SAVE command or make additional changes.

If TM_SAVE_ACTION is used as a return value, all the requests associated with the SAVE command (except TM_PRE_SAVE and TM_POST_SAVE) are completed, but this processing is not used in the standard models.

Table 32. Slice processing for the CLOSE *command.*

Slices	Typical Processing
TM_POST_CLOSE	Processing is identical to that of TM_POST_SAVE described under SAVE.
TM_POST_SAVE1	Described under SAVE, but no save cursor will exist
TM_POST_SAVE2	Described under SAVE

CONTINUE

Fetches the next set of information from the database

```
int sm_tm_command ("CONTINUE [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CONTINUE` fetches the next set of information from the database. If there are no additional rows, this command has no effect.

`CONTINUE` does not set the transaction mode but requires `view` or `update` mode. A partial `CONTINUE` command is also permitted in `new` mode.

Push buttons and menu selections for the `CONTINUE` command may choose to set the Class property to `continue_button`. By default, `continue_button` is active in `view` and `update` modes.

If your screen has multiple table views, the transaction manager issues a DBMS CONTINUE for the specified table view and any table views linked to it via server links. This displays the next set of rows for that server view. Then SELECT or VIEW processing is done for any additional child table views.

If your screen has multiple table views and you want to fetch data for only one table view, it is suggested that you use FETCH instead of CONTINUE.

warning about continuation files

If the setting of the Fetch Directions property, as discussed in the CONTINUE_DOWN command, permits the CONTINUE_DOWN command to be executed, the data displayed by this command for the specified server view may come from a continuation file. The warnings for CONTINUE_DOWN then apply.

Sequence

Use CONTINUE after SELECT or VIEW which generate a database query and display the first set of query results.

Requests

The following requests can be generated by the CONTINUE command to ascertain whether the changes from the previous command have been saved and, if desired, to discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

The following requests can also be generated:

- TM_FETCH (described under FETCH)
- TM_PRE_SELECT (described under SELECT)
- TM_SELECT (described under SELECT)
- TM_POST_SELECT (described under SELECT)
- TM_PRE_VIEW (described under VIEW)
- TM_VIEW (described under VIEW)

- `TM_POST_VIEW` (described under `VIEW`)

If `TM_VIEW` or `TM_SELECT` for a parent table view returns no data, `TM_CLEAR` requests are generated for all subordinate table views, but not for table views at the same level of the tree. `TM_CLEAR` requests are described under `CLEAR`.

CONTINUE_BOTTOM

Fetches the last set of rows from the file

```
int sm_tm_command ("CONTINUE_BOTTOM [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CONTINUE_BOTTOM` fetches the last set of rows from the file. The availability of this command is dependent on the setting of the Fetch Directions property for the server view or screen. If the Fetch Directions property is set to `Up/Down–all modes`, this command is available in `update` or `view` mode. If the Fetch Directions property is set to `Up/Down–view mode`, this command is available only in `view` mode. Otherwise, an error is generated. For more information on setting the Fetch Directions property, refer to page 377.

If your screen has multiple table views, the transaction manager issues a `DBMS CONTINUE_BOTTOM` for the specified table view and any table views linked to it

via server links. This displays the last set of rows for that server view. Then SELECT or VIEW processing is done for any additional child table views.

You should be aware that the data displayed with this command is from a continuation file. It is not re-fetched from the database. Therefore, any updates made to the data in this server view either by you, or by another user, are not displayed. In order to display those updates, you must again fetch the data from the database with a VIEW or SELECT command.

The advantage of using JAM's continuation file is that it prevents having shared locks on data. However, if the Fetch Directions property is set to Up/Down–all modes, you are responsible for implementing the necessary locking scheme for concurrent users. For more information on using the Version Column property to implement optimistic locking, refer to page 370.

If you want to use the database engine's facilities for non-sequential scrolling, you need to edit the transaction model.

Push buttons and menu selections for the CONTINUE_BOTTOM command may choose to set the Class property to continue_button which activates the option only in view and update modes or to continue_view_button which activates the option only in view mode.

Sequence

Use CONTINUE_BOTTOM after SELECT or VIEW which generate a database query and display the first set of query results or after any other CONTINUE command.

Requests

The following request is generated by the CONTINUE_BOTTOM command:

Request	Traversal	Typical Processing
TM_CONTINUE_BOTTOM	The table views in the specified server view	See below

Table 33. Transaction manager request and slice processing for the `CONTINUE_BOTTOM` *command.*

Slices	Typical Processing
`TM_CONTINUE_BOTTOM`	A select cursor must have been set up for the server view encompassing the current table view or nothing more is done. Calls `sm_tm_continuation_availability` to check if the command is available. If not, an error is issued. On entry, `TM_OCC_COUNT` specifies the maximum number of occurrences to be fetched. If `TM_OCC_COUNT` is zero on entry, it means that there is no explicit limit being imposed. The `TM_OCC` member on entry specifies the first occurrence to be fetched into. `TM_OCC_COUNT` is then zeroed. (At the end of this request, `TM_SEL_CHECK` sets it to contain the number of rows fetched.) The data is fetched. `TM_SEL_CHECK` is pushed onto the event stack to report the number of rows fetched.
`TM_SEL_CHECK`	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to `TM_OCC_COUNT`.

The following requests can be generated by the `CONTINUE_BOTTOM` command to ascertain if the changes from the previous command have been saved and, if desired, to discard those changes:

- `TM_PRE_CLOSE` (described under `CLOSE`)
- `TM_CLOSE` (described under `CLOSE`)
- `TM_QUERY` (described under `CLOSE`)
- `TM_DISCARD` (described under `CLOSE`)
- `TM_POST_CLOSE` (described under `CLOSE`)

The following requests can also be generated for any child table views:

- `TM_PRE_SELECT` (described under `SELECT`)

- TM_SELECT (described under SELECT)
- TM_POST_SELECT (described under SELECT)
- TM_PRE_VIEW (described under VIEW)
- TM_VIEW (described under VIEW)
- TM_POST_VIEW (described under VIEW)

If TM_VIEW or TM_SELECT for a parent table view returns no data, TM_CLEAR requests are generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

CONTINUE_DOWN

Fetches the next set of rows from the file

```
int sm_tm_command ("CONTINUE_DOWN [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CONTINUE_DOWN` fetches the next set of rows from the file. Note that even though the commands `CONTINUE_DOWN` and `CONTINUE` both display the next set of data, `CONTINUE_DOWN` generates different a request than `CONTINUE`.

The availability of `CONTINUE_DOWN` is dependent on the setting of the Fetch Directions property for the server view or screen. If the Fetch Directions property is set to `Up/Down–all modes`, this command is available in `update` or `view` mode. If the Fetch Directions property is set to `Up/Down–view mode`, this command is available only in `view` mode. Otherwise, an error is generated. For more information on setting the Fetch Directions property, refer to page 377.

If your screen has multiple table views, the transaction manager issues a DBMS CONTINUE_DOWN for the specified table view and any table views linked to it via server links. This displays the next set of rows for that server view. Then SELECT or VIEW processing is done for any additional child table views.

You should be aware that the data displayed with this command is from a continuation file. It is not re-fetched from the database. Therefore, any updates made to the data in this server view either by you, or by another user, are not displayed. In order to display those updates, you must again fetch the data from the database with a VIEW or SELECT command.

The advantage of using JAM's continuation file is that it prevents having shared locks on data. However, if the Fetch Directions property is set to Up/Down–all modes, you are responsible for implementing the necessary locking scheme for concurrent users. For more information on using the Version Column property to implement optimistic locking, refer to page 370.

If you want to use the database engine's facilities for non-sequential scrolling, you need to edit the transaction model.

Push buttons and menu selections for the CONTINUE_DOWN command may choose to set the Class property to continue_button which activates the option only in view and update modes or to continue_view_button which activates the option only in view mode.

Sequence

Use CONTINUE_DOWN after SELECT or VIEW which generate a database query and display the first set of query results or after any other CONTINUE command.

Requests

The following request is generated by the CONTINUE_DOWN command:

Request	Traversal	Typical Processing
TM_CONTINUE_DOWN	The table views in the specified server view	See below

Table 34. Transaction manager request and slice processing for the `CONTINUE_DOWN` *command.*

Slices	Typical Processing
`TM_CONTINUE_DOWN`	A select cursor must have been set up for the server view encompassing the current table view or nothing more is done. Calls `sm_tm_continuation_availability` to check if the command is available. If not, an error is issued. On entry, `TM_OCC_COUNT` specifies the maximum number of occurrences to be fetched. If `TM_OCC_COUNT` is zero on entry, it means that there is no explicit limit being imposed. The `TM_OCC` member on entry specifies the first occurrence to be fetched into. `TM_OCC_COUNT` is then zeroed. (At the end of this request, `TM_SEL_CHECK` sets it to contain the number of rows fetched.) The data is fetched. `TM_SEL_CHECK` is pushed onto the event stack to report the number of rows fetched.
`TM_SEL_CHECK`	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to `TM_OCC_COUNT`.

The following requests can be generated by the `CONTINUE_DOWN` command to ascertain if the changes from the previous command have been saved and, if desired, to discard those changes:

- `TM_PRE_CLOSE` (described under `CLOSE`)
- `TM_CLOSE` (described under `CLOSE`)
- `TM_QUERY` (described under `CLOSE`)
- `TM_DISCARD` (described under `CLOSE`)
- `TM_POST_CLOSE` (described under `CLOSE`)

The following requests can also be generated for any child table views:

- `TM_PRE_SELECT` (described under `SELECT`)

- TM_SELECT (described under SELECT)
- TM_POST_SELECT (described under SELECT)
- TM_PRE_VIEW (described under VIEW)
- TM_VIEW (described under VIEW)
- TM_POST_VIEW (described under VIEW)

If TM_VIEW or TM_SELECT for a parent table view returns no data, TM_CLEAR requests are generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

CONTINUE_TOP

Fetches the first set of rows from the file

```
int sm_tm_command ("CONTINUE_TOP [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CONTINUE_TOP` fetches the first set of rows from the file. The availability of this command is dependent on the setting of the Fetch Directions property for the server view or screen. If the Fetch Directions property is set to `Up/Down–all modes`, this command is available in `update` or `view` mode. If the Fetch Directions property is set to `Up/Down–view mode`, this command is available only in `view` mode. Otherwise, an error is generated. For more information on setting the Fetch Directions property, refer to page 377.

If your screen has multiple table views, the transaction manager issues a `DBMS CONTINUE_TOP` for the specified table view and any table views linked to it via

server links. This displays the first set of rows for that server view. Then SELECT or VIEW processing is done for any additional child table views.

You should be aware that the data displayed with this command is from a continuation file. It is not re-fetched from the database. Therefore, any updates made to the data in this server view either by you, or by another user, are not displayed. In order to display those updates, you must again fetch the data from the database with a VIEW or SELECT command.

The advantage of using JAM's continuation file is that it prevents having shared locks on data. However, if the Fetch Directions property is set to Up/Down–all modes, you are responsible for implementing the necessary locking scheme for concurrent users. For more information on using the Version Column property to implement optimistic locking, refer to page 370.

If you want to use the database engine's facilities for non-sequential scrolling, you need to edit the transaction model.

Push buttons and menu selections for the CONTINUE_TOP command may choose to set the Class property to continue_button which activates the option only in view and update modes or to continue_view_button which activates the option only in view mode.

Sequence

Use CONTINUE_TOP after SELECT or VIEW which generate a database query and display the first set of query results or after any other CONTINUE command.

Requests

The following request is generated by the CONTINUE_TOP command:

Request	Traversal	Typical Processing
TM_CONTINUE_TOP	The table views in the specified server view	See below

Table 35. Transaction manager request and slice processing for the `CONTINUE_TOP` *command.*

Slices	Typical Processing
`TM_CONTINUE_TOP`	A select cursor must have been set up for the server view encompassing the current table view or nothing more is done. Calls `sm_tm_continuation_availability` to check if the command is available. If not, an error is issued. On entry, `TM_OCC_COUNT` specifies the maximum number of occurrences to be fetched. If `TM_OCC_COUNT` is zero on entry, it means that there is no explicit limit being imposed. The `TM_OCC` member on entry specifies the first occurrence to be fetched into. `TM_OCC_COUNT` is then zeroed. (At the end of this request, `TM_SEL_CHECK` sets it to contain the number of rows fetched.) The data is fetched. `TM_SEL_CHECK` is pushed onto the event stack to report the number of rows fetched.
`TM_SEL_CHECK`	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to `TM_OCC_COUNT`.

The following requests can be generated by the `CONTINUE_TOP` command to ascertain if the changes from the previous command have been saved and, if desired, to discard those changes:

- `TM_PRE_CLOSE` (described under `CLOSE`)
- `TM_CLOSE` (described under `CLOSE`)
- `TM_QUERY` (described under `CLOSE`)
- `TM_DISCARD` (described under `CLOSE`)
- `TM_POST_CLOSE` (described under `CLOSE`)

The following requests can also be generated for any child table views:

- TM_PRE_SELECT (described under SELECT)
- TM_SELECT (described under SELECT)
- TM_POST_SELECT (described under SELECT)
- TM_PRE_VIEW (described under VIEW)
- TM_VIEW (described under VIEW)
- TM_POST_VIEW (described under VIEW)

If TM_VIEW or TM_SELECT for a parent table view returns no data, TM_CLEAR requests are generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

CONTINUE_UP

Fetches the previous set of rows from the file

```
int sm_tm_command ("CONTINUE_UP [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CONTINUE_UP` fetches the previous set of rows from the file. The availability of this command is dependent on the setting of the Fetch Directions property for the server view or screen. If the Fetch Directions property is set to `Up/Down–all modes`, this command is available in `update` or `view` mode. If the Fetch Directions property is set to `Up/Down–view mode`, this command is available only in `view` mode. Otherwise, an error is generated. For more information on setting the Fetch Directions property, refer to page 377.

If your screen has multiple table views, the transaction manager issues a `DBMS CONTINUE_UP` for the specified table view and any table views linked to it via

server links. This displays the previous set of rows for that server view. Then SELECT or VIEW processing is done for any additional child table views.

You should be aware that the data displayed with this command is from a continuation file. It is not re-fetched from the database. Therefore, any updates made to the data in this server view either by you, or by another user, are not displayed. In order to display those updates, you must again fetch the data from the database with a VIEW or SELECT command.

The advantage of using JAM's continuation file is that it prevents having shared locks on data. However, if the Fetch Directions property is set to Up/Down–all modes, you are responsible for implementing the necessary locking scheme for concurrent users. For more information on using the Version Column property to implement optimistic locking, refer to page 370.

If you want to use the database engine's facilities for non-sequential scrolling, you need to edit the transaction model.

Push buttons and menu selections for the CONTINUE_UP command may choose to set the Class property to continue_button which activates the option only in view and update modes or to continue_view_button which activates the option only in view mode.

Sequence

Use CONTINUE_UP after SELECT or VIEW which generate a database query and display the first set of query results or after any other CONTINUE command.

Requests

The following request is generated by the CONTINUE_UP command:

Request	Traversal	Typical Processing
TM_CONTINUE_UP	The table views in the specified server view	See below

Table 36. Transaction manager request and slice processing for the `CONTINUE_UP` *command.*

Slices	Typical Processing
`TM_CONTINUE_UP`	A select cursor must have been set up for the server view encompassing the current table view or nothing more is done. Calls `sm_tm_continuation_availability` to check if the command is available. If not, an error is issued. On entry, `TM_OCC_COUNT` specifies the maximum number of occurrences to be fetched. If `TM_OCC_COUNT` is zero on entry, it means that there is no explicit limit being imposed. The `TM_OCC` member on entry specifies the first occurrence to be fetched into. `TM_OCC_COUNT` is then zeroed. (At the end of this request, `TM_SEL_CHECK` sets it to contain the number of rows fetched.) The data is fetched. `TM_SEL_CHECK` is pushed onto the event stack to report the number of rows fetched.
`TM_SEL_CHECK`	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to `TM_OCC_COUNT`.

The following requests can be generated by the `CONTINUE_UP` command to ascertain if the changes from the previous command have been saved and, if desired, to discard those changes:

- `TM_PRE_CLOSE` (described under `CLOSE`)
- `TM_CLOSE` (described under `CLOSE`)
- `TM_QUERY` (described under `CLOSE`)
- `TM_DISCARD` (described under `CLOSE`)
- `TM_POST_CLOSE` (described under `CLOSE`)

The following requests can also be generated for any child table views:

- `TM_PRE_SELECT` (described under `SELECT`)

- TM_SELECT (described under SELECT)
- TM_POST_SELECT (described under SELECT)
- TM_PRE_VIEW (described under VIEW)
- TM_VIEW (described under VIEW)
- TM_POST_VIEW (described under VIEW)

If TM_VIEW or TM_SELECT for a parent table view returns no data, TM_CLEAR requests are generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

COPY

Duplicates the data on the screen so it can be edited

```
int sm_tm_command (”COPY”);
```

Description

COPY copies the data on the screen for use in the next insertion.

After you select COPY, the following steps occur:

1. If you have made changes in the table views on which this command operates in a previous NEW, COPY, COPY_FOR_UPDATE, or SELECT, you are asked if you want to discard your changes. If you press OK, the changes are discarded; however, the data remains visible and is treated as though you had just typed it in after a NEW command. If you press Cancel, you will be returned to the screen so you can save your changes.

2. The data currently displayed on the screen is copied.

3. The transaction mode is set to new. By default, this mode clears all the protection bits in updatable table views to reflect that data entry is available in those widgets.

4. Edit the data as much as you wish. Select SAVE to insert the data into the database. If you select SAVE without changing any data, then the transaction manager generates an INSERT statement for the duplicate data. Depending on the engine, this could result in a duplicate entry or in an engine error.

Push buttons and menu selections for the COPY command may choose to set the Class property to copy_button. By default, copy_button is active in all transaction modes.

Sequence

COPY is available after you enter new data using NEW and SAVE. It is also available after SELECT or VIEW which display data on the screen. Select SAVE after you finish your edits.

Requests

The following requests can be generated by the COPY command to ascertain whether the changes from the previous command have been saved and, if desired, to discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

Since no TM_DISCARD request is made for the COPY command, the discard flag used in TM_POST_SAVE1 will not be set.

Table 37. Transaction manager requests for the COPY *command.*

Request	Traversal	Typical Processing
TM_PRE_COPY	By table/server view from the specified table view	Do nothing
TM_COPY	By table/server view from the specified table view	Do nothing (sm_bi_init_copy is called for the table view by the transaction manager after this request)
TM_POST_COPY	By table/server view from the specified table view	Do nothing

See Also

NEW

COPY_FOR_UPDATE

Changes the transaction manager to update mode

```
int sm_tm_command (”COPY_FOR_UPDATE”);
```

Description

COPY_FOR_UPDATE changes the current mode to update. This allows the data currently displayed on the screen to be modified, as though it had been fetched from the database. After you select COPY_FOR_UPDATE, the transaction manager initializes before image processing.

If you edit the data and select SAVE, the transaction manager will generate statements as if the data now on the screen had come from a SELECT command. If corresponding data is not in the database, the results may not be what you expect.

Push buttons and menu selections for the COPY_FOR_UPDATE command may choose to set the Class property to continue_button since, by default, continue_button is active in view or update modes.

Sequence

COPY_FOR_UPDATE is available from any mode. Select SAVE after you finish your edits.

Requests

The following requests can be generated by the COPY_FOR_UPDATE command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

Table 38. Transaction manager requests for the `COPY_FOR_UPDATE` *command.*

Request	Traversal	Typical Processing
`TM_PRE_COPY_FOR_UPDATE`	By table/server view from the specified table view	Do nothing
`TM_COPY_FOR_UPDATE`	By table/server view from the specified table view	Do nothing (`sm_bi_initialize` and `sm_bi_copy` are called for the table view by the transaction manager after this request)
`TM_POST_COPY_FOR_UPDATE`	By table/server view from the specified table view	Do nothing

COPY_FOR_VIEW

Changes the transaction manager to view mode

```
int sm_tm_command (”COPY_FOR_VIEW”);
```

Description

COPY_FOR_VIEW makes view the current mode. After you select COPY_FOR_VIEW, the transaction manager disables before image processing. Changes to the data currently on the screen no longer generate updates to the database with a SAVE command.

Push buttons and menu selections for the COPY_FOR_UPDATE command may choose to set the Class property to continue_button. By default, continue_button is active in view or update modes.

Sequence

COPY_FOR_VIEW is available after any command.

Requests

The following requests can be generated by the COPY_FOR_VIEW command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

Table 39. Transaction manager requests for the `COPY_FOR_VIEW` *command.*

Request	Traversal	Typical Processing
`TM_PRE_COPY_FOR_VIEW`	By table/server view from the specified table view	Do nothing
`TM_COPY_FOR_VIEW`	By table/server view from the specified table view	Do nothing (`sm_bi_suppress` is called for the table view by the transaction manager after this request)
`TM_POST_COPY_FOR_VIEW`	By table/server view from the specified table view	Do nothing

FETCH

Fetches the next set of data from the database

```
int sm_tm_command ("FETCH [ table-view-name [ { FETCH_SIMPLE | FETCH_SPECIAL } ] ]");
```

table-view-name	The name of a table view in the current transaction. This parameter is case sensitive. If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0. If *table_view_name* is not specified, the command is applied for the root table view.
`FETCH_SIMPLE`	Start the fetch with the first occurrence. The number of rows fetched depends on the size of the arrays. This is the parameter used if neither `FETCH_SIMPLE` nor `FETCH_SPECIAL` is specified, or if no table view name is specified.
`FETCH_SPECIAL`	Allows you to override the occurrence number and the size of the array. To use the `FETCH_SPECIAL` parameter, you must set the value of `TM_OCC` and `TM_OCC_COUNT` with `sm_tm_iset` before calling this command. When `FETCH_SPECIAL` is specified, `TM_OCC` is consulted for the start position and `TM_OCC_COUNT` is consulted for the count.

Description

`FETCH` fetches the next set of rows for the specified table view.

If your screen has multiple table views and you want to fetch data for all of them at the same time, it is suggested that you use `CONTINUE` instead of `FETCH`.

Push buttons and menu selections for the `FETCH` command may choose to set the Class property to `continue_button`. By default, `continue_button` is active in `view` and `update` modes.

Sequence

This command is available after `SELECT` or `VIEW`, both of which generate a database query and display the first set of query results.

Requests

The following requests can be generated by the FETCH command:

Request	Traversal	Typical Processing
TM_FETCH	No tree traversal, since performed only for the specified table view	Slices: TM_FETCH, TM_SEL_CHECK

Table 40. Slice processing for the FETCH *command.*

Slices	Typical Processing
TM_FETCH	A select cursor must have been set up for the server view encompassing the current table view or nothing more is done. On entry, TM_OCC_COUNT specifies the maximum number of occurrences to be fetched. If TM_OCC_COUNT is zero on entry, it means that there is no explicit limit being imposed. The TM_OCC member on entry specifies the first occurrence to be fetched into. TM_OCC_COUNT is then zeroed. (At the end of this event, TM_SEL_CHECK sets it to contain the number of rows fetched.) The data is fetched. TM_SEL_CHECK is pushed onto the event stack to report the number of rows fetched.
TM_SEL_CHECK	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to TM_OCC_COUNT. Give up the select cursor if there are no more rows unless a continuation file is in use.

FINISH

Closes the current transaction manager transaction

```
int sm_tm_command (”FINISH”);
```

Description

`FINISH` contains the screen exit processing needed by the transaction manager and is called automatically on screen exit. If you use only the default transaction manager transaction on your screen, you would not need to explicitly call this command.

As part of its processing, `FINISH` closes the current transaction, which has been set with the `START` or `CHANGE` commands. In cases where you initiate multiple transactions on the same screen by calling the `START` command, you would need to call the `FINISH` command to close those transactions before exiting the screen.

The `FINISH` command is called after the named screen exit function and after the default screen function. After `FINISH`, the transaction manager data structures for what had been the current transaction no longer exist.

Requests

The following request is generated by the `FINISH` command:

Request	Traversal	Typical Processing
`TM_FINISH`	By table/server view from the root table view. Done both for hook functions and the transaction model.	Slice: `TM_FINISH`

Table 41. Slice processing for the `FINISH` *command.*

Slices	Typical Processing
`TM_FINISH`	Give up the save cursor (if it is in use) and the select cursor for the server view encompassing the current table view (if it is in use). For engines where giving up a cursor involves closing the cursor, the return value will be `TM_CHECK`. Give up data areas allocated to this transaction, but not areas that are allocated for the transaction model, since there may be other transactions that are still active.

Example

For an example which also uses the `START` command, refer to page 455.

FORCE_CLOSE

Unconditionally discards the changes to the screen

```
int sm_tm_command ("FORCE_CLOSE [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`FORCE_CLOSE` discards the changes to the screen without a query message. If a table view is not specified, it sets the transaction mode to `initial`.

Push buttons and menu selections for the `FORCE_CLOSE` command may choose to set the Class property to `close_button`. By default, `close_button` is active in all but `initial` mode.

Sequence

This command is useful after `SELECT`, `NEW` or `COPY` in order to discard your changes.

Requests

The following requests can be generated by the FORCE_CLOSE command to discard changes that may have been made to the screen.

Table 42. Transaction manager requests for the FORCE_CLOSE *command.*

Request	Traversal	Typical Processing
TM_PRE_CLOSE	By table/server view from the specified table view	Note that SAVE/CLOSE processing is beginning. Identical processing is performed for TM_PRE_SAVE.
TM_DISCARD	By table/server view from the specified table view	Set a discard flag, consulted by TM_POST_SAVE1
TM_POST_CLOSE	By table/server view from the specified table view	Slices: TM_POST_CLOSE, TM_POST_SAVE1, TM_POST_SAVE2 For some engines, the processing in TM_POST_SAVE1 may suggest a change to initial mode at the end of this request.

Table 43. Slice processing for the FORCE_CLOSE *command.*

Slices	Typical Processing
TM_POST_CLOSE	Processing is identical to that of TM_POST_SAVE described under SAVE.
TM_POST_SAVE1	Described under SAVE, but no save cursor will exist
TM_POST_SAVE2	Described under SAVE

NEW

Prepares screen for data entry

```
int sm_tm_command ("NEW [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. A table view may only be specified if the mode has already been set to `new`. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`NEW` clears each field and prepares it for data entry. To insert data successfully, all the fields in a table view that are participating in the SQL `INSERT` statement need to have the same number of occurrences.

After you select `NEW`, the following steps occur:

1. If you have made changes in the table views on which this command operates in a previous `NEW`, `COPY` or `SELECT`, you are asked if you want to discard your changes. If you press OK, the changes are discarded and fields in the specified

table views are cleared. If you press Cancel, you will be returned to the screen so you can save your changes. You must then select NEW again.

2. The fields are cleared of all previous values.
3. The transaction mode is set to new. By default, this mode clears all the protection bits in updatable table views to reflect that data entry is available in those widgets.
4. Before image processing for the screen is enabled. Any changes made to the screen following this step can then be processed using SAVE.

Push buttons and menu selections for the NEW command may choose to set the Class property to new_button. By default, new_button is active in initial and view modes.

Sequence

To save the additions, select SAVE as the next transaction command. To discard the additions, select CLOSE or FORCE_CLOSE.

If you are entering a series of rows, COPY copies the data on a screen so it can then be edited, without having to enter the data again.

Requests

The following requests can be generated by the NEW command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

Table 44. Transaction manager requests for the `NEW` *command.*

Request	Traversal	Typical Processing
`TM_PRE_NEW`	By table/server view from the specified table view	Do nothing
`TM_NEW`	By table/server view from the specified table view	Do nothing (`sm_bi_initialize` and `sm_bi_copy` are called for the table view by the transaction manager after this request)
`TM_POST_NEW`	By table/server view from the specified table view	Do nothing

REFRESH

Refreshes the screen in order to update the style and class settings

```
int sm_tm_command (”REFRESH”);
```

Description

`REFRESH` reapplies the styles and classes for the current mode.

Requests

There are no requests generated by the `REFRESH` command.

SAVE

Saves the changes made on the screen to the database

```
int sm_tm_command ("SAVE [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`SAVE` compares the current screen to the before image data and generates the necessary statements needed to update the database.

After you select `SAVE`, the following steps occur:

1. The transaction manager checks to see that the mode is not `initial` or `view`.
2. For engines requiring it, the transaction model starts a database transaction.

3. The transaction model calls the SQL generator to execute the necessary statements according to the changes that were entered on the screen. Statements can only be generated for updatable table views.

   Note that some database engines discard the select set when a commit or rollback is performed. For those engines, the standard transaction models give up the select cursor after a commit or rollback.

4. If an error is encountered, the database transaction is rolled back. If no errors are reported and the SAVE command has been specified as a full command, the transaction model commits the database transaction.

Push buttons and menu selections for the SAVE command may choose to set the Class property to `save_button`. By default, `save_button` is active in `new` and `update` modes.

primary key changes

The transaction manager is aware of any primary key changes. If the primary key is updated, the standard transaction models delete the row containing the old value of the primary key and insert a row contains the new value of the primary key. If the primary key is cleared, the standard transaction models delete the row corresponding to the cleared key fields.

Requests

The following requests can be generated by the SAVE command:

- `TM_PRE_SAVE`
- `TM_SAVE`
- `TM_DELETE`
- `TM_UPDATE`
- `TM_INSERT`
- `TM_POST_SAVE`

Table 45. Transaction manager requests for the SAVE *command.*

Request	Traversal	Typical Processing
TM_PRE_SAVE	By table/server view from the specified table view	Note that SAVE/CLOSE processing is beginning. Set the reuse cursor flag to 0. (Processing is identical for TM_PRE_CLOSE.)
TM_SAVE	By table/server view from the specified table view	Do nothing
TM_DELETE	Modify table views in the order specified in the link properties, with one request in each table view for each row in that table view that has changed	Slices: TM_DELETE, TM_GET_SAVE_CURSOR, TM_DELETE_DECLARE, TM_DELETE_EXEC
TM_UPDATE	Modify table views in the order specified in the link properties, with one request in each table view for each row in that table view that has changed	Slices: TM_UPDATE, TM_GET_SAVE_CURSOR, TM_UPDATE_DECLARE, TM_UPDATE_EXEC
TM_INSERT	Modify table views in the order specified in the link properties, with one request in each table view for each row in that table view that has changed	Slices: TM_INSERT, TM_GET_SAVE_CURSOR, TM_INSERT_DECLARE, TM_INSERT_EXEC
TM_POST_SAVE	By table/server view from the specified table view	Slices: TM_POST_SAVE, TM_POST_SAVE1, TM_GIVE_UP_SAVE_CURSOR, TM_POST_SAVE2

Table 46. Slice processing for the SAVE *command.*

Slices	Typical Processing
TM_GET_SAVE_CURSOR	If a name does not exist for the save cursor, generate it. For some engines, special processing may be necessary to begin a transaction.

Slices	Typical Processing
TM_DELETE	Find out what type of change was made to the current occurrence by checking the return code from sm_bi_compare. If the return code is BI_KEY_CHANGED, BI_KEY_NULLED, or BI_DELETED, push the TM_GET_SAVE_CURSOR, TM_DELETE_DECLARE and TM_DELETE_EXEC events onto the stack.
TM_DELETE_DECLARE	For some engines, give up any select cursor relating to this table view. Call dm_exec_sql to declare the save cursor for this deletion, unless the occurrence is part of an array and previously generated SQL is being reused.
TM_DELETE_EXEC	Call dm_exec_sql to execute the save cursor for this deletion. The return value is TM_CHECK_ONE_ROW which tests that only one row was deleted.
TM_UPDATE	Find out what type of change was made to the current occurrence by checking the return code from sm_bi_compare. If the return code is BI_UPDATED, push the TM_GET_SAVE_CURSOR, TM_UPDATE_DECLARE and TM_UPDATE_EXEC events onto the stack.
TM_UPDATE_DECLARE	For some engines, give up any select cursor relating to this table view. Call dm_exec_sql to declare the save cursor for this update, unless the occurrence is part of an array and previously generated SQL is being reused.
TM_UPDATE_EXEC	Call dm_exec_sql to execute the save cursor for this update. The return value is TM_CHECK_ONE_ROW which tests that only one row was updated.
TM_INSERT	Find out what type of change was made to the current occurrence by checking the return code from sm_bi_compare. If the return code is BI_KEY_CHANGED or BI_INSERTED, push the TM_GET_SAVE_CURSOR, TM_INSERT_DECLARE and TM_INSERT_EXEC events onto the stack.

Slices	Typical Processing
TM_INSERT_DECLARE	For some engines, give up any select cursor relating to this table view. Call dm_exec_sql to declare the cursor for this insertion, unless the occurrence is part of an array and previously generated SQL is being reused.
TM_INSERT_EXEC	Call dm_exec_sql to execute the save cursor for this insertion. The return value is TM_CHECK_ONE_ROW which tests that only one row was inserted.
TM_POST_SAVE	If this is the first TM_POST_SAVE event since the last TM_PRE_CLOSE or TM_PRE_SAVE, push the TM_POST_SAVE1 event on the stack. Otherwise, if the saving worked flag was set to 1 in TM_POST_SAVE1 processing, push the TM_POST_SAVE2 event on the stack.
TM_POST_SAVE1	The existence of a save cursor indicates that SQL statements were executed so the saving worked flag is set to 1. If there is a save cursor and if TM_STATUS is equal to 0 (indicating that the statements executed successfully) and if TM_FULL is equal to 1 (indicating a full SAVE command), a DBMS COMMIT is executed with a return code of TM_CHECK. If there is a save cursor and if TM_STATUS is non–zero (indicating that the statements failed), a DBMS ROLLBACK is executed with a return code of TM_CHECK. For rollbacks, the saving worked flag is reset to 0 since no changes were actually made. If there is no save cursor, the saving worked flag is set to 1 if, and only if, the discard flag is set (see TM_DISCARD). If the saving worked flag is set to 1, the TM_POST_SAVE2 event is pushed, to perform that processing for this first table view. If there was a save cursor, the TM_GIVE_UP_SAVE_CURSOR event is pushed, to give up the save cursor.

Slices	Typical Processing
	Note: Some engines discard the select sets when commits and rollbacks are performed. For those engines, TM_VALUE is set to suggest changing to initial mode at the end of the TM_POST_SAVE request if a commit or rollback was done.
TM_GIVE_UP_SAVE_CURSOR	Give up the save cursor.
TM_POST_SAVE2	Call sm_bi_initialize. Set TM_OCC to 1 and TM_OCC_COUNT to –1; then, call sm_bi_copy.

SELECT

Fetches data from the database to be updated

```
int sm_tm_command ("SELECT [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`SELECT` fetches data from the database so it can be modified. In order to successfully update data or insert new data, all the fields in a server view which are included in the select list need to have the same number of occurrences.

After you choose `SELECT`, the following steps occur:

1. If you have made changes in the table views on which this command operates in a previous `NEW`, `COPY`, `COPY_FOR_UPDATE`, or `SELECT`, you are asked if you want to discard your changes. If you press OK, the changes are discarded and

fields in the specified table views are cleared. If you press Cancel, you will be returned to the screen so you can save your changes.

2. The transaction mode is set to `update` unless a table view is specified and the mode is not initial mode. By default, update mode protects the primary key fields from data entry and sets the display attributes differently for key and non–key fields.

3. The screen displays the first set of data for all linked table views.

   When you choose `SELECT`, the standard transaction models have the SQL generator execute a SQL `SELECT` statement for the database table named in the root table view and any table views connected to it via a server link. Then, recursively, SQL `SELECT` statements are issued for the child table views having sequential links, and any table views connected to those child table views by server links.

4. The before image, or snapshot, of the screen is taken for the screen‘s updatable table views. An updatable table view must have its primary key fields on screen. Any changes made to the screen following this step can then be processed using a `SAVE` command.

Push buttons and menu selections for the `SELECT` command may choose to set the Class property to `view_button`. By default, `view_button` is active in `initial` or `view` modes.

using qbe

If you want to select a specific record or group of records, set the widget’s Use in Where property to Yes and the type of operator to be used in the `WHERE` clause. Then, in the transaction manager, choose `CLEAR` to clear the fields, enter a value in your query field, and then choose `SELECT`. The screen displays the specified information.

Sequence

To save the changes or additions made to the selected data, choose `SAVE` as the next transaction command.

To display the next row of information, choose `CONTINUE` as the next transaction command. If you have updated the data on the screen, a dialogue box asks if you want to discard your changes. If you press `OK`, the changes are discarded. If you press `Cancel`, you will be returned to the screen so you can save your changes.

To discard any changes you have made to the screen, choose `CLOSE` or `FORCE_CLOSE`.

Requests

The following requests can be generated by the SELECT command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

The SELECT command generates TM_CLEAR requests if TM_SELECT for a parent table view returns no data. In that case, TM_CLEAR is generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

Table 47. Transaction manager requests for the SELECT *command.*

Request	Traversal	Typical Processing
TM_PRE_SELECT	By table/server view from the specified table view	Do nothing
TM_SELECT	By table/server view from the specified table view	Slices: TM_SELECT, TM_GET_SEL_CURSOR, TM_PREPARE_CONTINUE, TM_SEL_GEN, TM_SEL_BUILD_PERFORM, TM_SEL_CHECK (sm_bi_initialize is called for the table view by the transaction manager after this request. If rows were fetched, sm_bi_copy is also called.)
TM_POST_SELECT	By table/server view from the specified table view	Do nothing

Table 48. Slice processing for the `SELECT` *command.*

Slices	Typical Processing
`TM_SELECT`	`TM_OCC_COUNT` is zeroed. At the end of processing for this request, it will contain the number of rows fetched (set, if at all, by `TM_SEL_CHECK`). If the table view is the first one in the current server view, push the `TM_GET_SEL_CURSOR` event (only if there is no select cursor already) and the `TM_PREPARE_CONTINUE`, `TM_SEL_GEN`, `TM_SEL_BUILD_PERFORM`, and `TM_SEL_CHECK` events onto the stack. If the table view is not the first one in the server view, nothing more is done for this request, and the number of rows fetched for this request is correctly reported as zero.
`TM_GET_SEL_CURSOR`	If a name does not exist for the JAM select cursor, generate it. (Depending on the engine, a JAM cursor may or may not correspond to a database cursor.)
`TM_PREPARE_CONTINUE`	If the select cursor does not already exist, a dummy `DECLARE CURSOR` command is issued. If `sm_tm_continuation_validity` reports that continuation file commands (like `CONTINUE_TOP`) are valid, `DBMS STORE FILE` is issued. If the function reports that those commands are invalid, `DBMS STORE` is issued.
`TM_SEL_GEN`	Generate data structures with `dm_gen_sql_info` that will be used in the `TM_SEL_BUILD_PERFORM` slice to build the SQL statements.
`TM_SEL_BUILD_PERFORM`	Build, and then (if there was no error in building) perform SQL `SELECT` (and other DBMS commands) with `dm_exec_sql`. Free the select info.

Slices	Typical Processing
TM_SEL_CHECK	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to TM_OCC_COUNT. Give up the select cursor if there are no more rows unless a continuation file is in use. (On engines where this means that the cursor is closed, the return code is TM_CHECK. Otherwise, the return code is TM_OK.)

If TM_SELECT for a parent table view returns no data, TM_CLEAR requests are generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

START

Initializes a new transaction tree

```
int sm_tm_command ("START transaction-name [ table-view-name [ table-view-scope ] ]");
```

transaction-name — The name of a transaction to be used for this screen.

table-view-name — The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope — One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`START` initiates a transaction manager transaction and makes it the current transaction. The mode is set to `initial`. Since this command is called automatically on screen entry whenever a root table view can be determined, the only situation where you would need to call this command is if you are using more than one transaction manager transaction on the same screen.

A transaction manager transaction must be in progress in order to call commands.

Note that in the JAM screen entry events, the unnamed JPL procedure is called *before* the START command. For this reason, transaction manager commands cannot be invoked in the unnamed procedure. After the START command is called, JAM then calls the default screen function and the named screen entry function.

Sequence

Once a transaction has been initiated with the START command, you can make it the current transaction using the CHANGE command.

Requests

The following requests can be generated by the START command:

Request	Traversal	Typical Processing
TM_START	By table/server view from the specified table view. Done both for hook functions and the transaction model.	Do nothing

Example

The following example illustrates the use of the START, CHANGE, and FINISH commands in order to execute transaction manager commands on an unlinked table view.

In this example, pricecats is the unlinked table view. The procedure start_new_tran first finds the name of the current transaction, starts a new transaction for the pricecats table view, and then changes to that transaction in order to execute a VIEW command. The procedure change_to_main changes back to the original transaction in order to execute transaction manager commands on those table views. The procedure change_to_new_tran changes to the new transaction in order to execute transaction manager commands on the pricecats table view. The procedure exit is set as the value of the Screen Exit property.

```
# JPL Procedures:

vars main_tran(31)

proc start_new_tran
main_tran=sm_tm_pinquire(TM_TRAN_NAME)
call sm_tm_command(”START price_tran pricecats”)
call sm_tm_command(”CHANGE price_tran”)
call sm_tm_command(”VIEW”)
return
```

```
proc change_to_main
call sm_tm_command(”CHANGE :main_tran”)
return

proc change_to_new_tran
call sm_tm_command(”CHANGE price_tran”)
return

# Screen exit property set to the following procedure.
proc exit(screen, flags)
if (flags & K_EXIT)
{
   call sm_tm_command(”CHANGE price_tran”)
   call sm_tm_command(”FINISH”)
   call sm_tm_command(”CHANGE :main_tran”)
   call sm_tm_command(”FINISH”)
}
return
```

VIEW

Fetches data from the database for display purposes

```
int sm_tm_command ("VIEW [ table-view-name [ table-view-scope ] ]");
```

table-view-name

The name of a table view in the current transaction. This parameter is case sensitive.

If *table_view_name* is specified, the command is applied according to the *scope* parameter. Since the entire table view tree may not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *table_view_name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

table-view-scope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`VIEW` fetches data from the database for viewing purposes only.

When `VIEW` is selected the following steps occur:

1. If you have made changes in the table views on which this command operates in a previous `NEW`, `COPY`, `COPY_FOR_UPDATE`, or `SELECT`, you are asked if you want to discard your changes. If you press OK, the changes are discarded and fields in the specified table views are cleared. If you press Cancel, you will be returned to the screen so you can save your changes.

2. The transaction mode is set to `view` unless a table view is specified. By default, view mode protects all fields from data entry.

3. The screen displays the first set of data for all linked table views.

   When you choose `VIEW`, the standard transaction models have the SQL generator execute a SQL `SELECT` statement for the database table named in the root table view and any table views connected to it via a server link. Then, recursively, SQL `SELECT` statements are issued for the child table views having sequential links, and any table views connected to those child table views by server links.

   If the query does not return any rows for the first server view, no data will be displayed for the remaining server views. (A query which successfully returns rows sets `TM_OCC_COUNT` as part of the `TM_SEL_CHECK` slice. When `TM_OCC_COUNT` is greater than 0, the query is generated for the next server view.)

Push buttons and menu selections for the `SELECT` command may choose to set the Class property to `view_button`. By default, `view_button` is active in `initial` or `view` modes.

using qbe

If you want to select a specific record or group of records, you need to set the Use in Where property to Yes and set the type of operator to be used in the `WHERE` clause. Then, in the transaction manager, choose `CLEAR` to clear the fields, enter a value in your query field, and then choose `VIEW`. The screen displays the specified information.

Sequence

To display additional data, choose any `CONTINUE` command.

Requests

The following requests can be generated by the `VIEW` command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

- `TM_PRE_CLOSE` (described under `CLOSE`)
- `TM_CLOSE` (described under `CLOSE`)
- `TM_QUERY` (described under `CLOSE`)
- `TM_DISCARD` (described under `CLOSE`)
- `TM_POST_CLOSE` (described under `CLOSE`)

The VIEW command generates TM_CLEAR requests if TM_VIEW for a parent table view returns no data. In that case, TM_CLEAR is generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

Table 49. Transaction manager requests for the VIEW *command.*

Request	Traversal	Typical Processing
TM_PRE_VIEW	By table/server view from the specified table view	Do nothing
TM_VIEW	By table/server view from the specified table view	Slices: TM_VIEW, TM_GET_SEL_CURSOR, TM_PREPARE_CONTINUE, TM_SEL_GEN , TM_SEL_BUILD_PERFORM, TM_SEL_CHECK (sm_bi_suppress is called for the table view by the transaction manager after this request.)
TM_POST_VIEW	By table/server view from the specified table view	Do nothing

Table 50. Slice processing for the VIEW *command.*

Slices	Typical Processing
TM_VIEW	TM_OCC_COUNT is zeroed. At the end of processing for this request, it will contain the number of rows fetched (set, if at all, by TM_SEL_CHECK). If the table view is the first one in the current server view, push the TM_GET_SEL_CURSOR event (only if there is no select cursor already) and the TM_PREPARE_CONTINUE, TM_SEL_GEN, TM_SEL_BUILD_PERFORM, and TM_SEL_CHECK events onto the stack. If the table view is not the first one in the server view, nothing more is done for this request, and the number of rows fetched for this request is correctly reported as zero.

Slices	Typical Processing
TM_GET_SEL_CURSOR	If a name does not exist for the JAM select cursor, generate it. (Depending on the engine, a JAM cursor may or may not correspond to a database cursor.)
TM_PREPARE_CONTINUE	If the select cursor does not already exist, a dummy DECLARE CURSOR command is issued. If sm_tm_continuation_validity reports that continuation file commands (like CONTINUE_TOP) are valid, DBMS STORE FILE is issued. If the function reports that those commands are invalid, DBMS STORE is issued.
TM_SEL_GEN	Generate data structures with dm_gen_sql_info that will be used in the TM_SEL_BUILD_PERFORM slice to build the SQL statements.
TM_SEL_BUILD_PERFORM	Build, and then (if there was no error in building) perform SQL SELECT (and other DBMS commands) with dm_exec_sql. Free the select info.
TM_SEL_CHECK	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to TM_OCC_COUNT. Give up the select cursor if there are no more rows unless a continuation file is in use.

If TM_VIEW for a parent table view returns no data, TM_CLEAR requests are generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

Transaction Manager Troubleshooting

This chapter lists some guidelines to use when creating transaction manager screens and lists the common errors along with the actions to take to correct them.

Guidelines for Creating Screens

In constructing an application screen, it is recommended that you observe the guidelines in the following sections.

Using Table Views

- If you are using more than one table view on the screen, you may want to copy the widgets for the root/parent table view first.

  In a master/detail screen, the primary key columns need to reside in the parent table view in order to display values and to use those values in SQL generation. By copying the widgets for the root/parent table views first, you do not need to edit the table view members.

- If you want to insert, update or delete information in a database table, the primary key columns for that table must be in the current table view or in a parent table view.

- The number of occurrences must be the same for all the members of a table view participating in SQL `INSERT`, `UPDATE`, and `DELETE` statements. Otherwise, the transaction manager generates an error.
- If you add a new widget to the screen and you want that widget to participate in SQL generation or transaction management, then the widget must be a member of a table view. For information on how to add members to a table view, refer Chapter 21 in the *Editors Guide*.
- A widget's protection properties can be changed at runtime corresponding to the class and style settings for each transaction mode. As a result, you may choose not to edit these properties in the screen editor or to edit the class and style settings instead.

  If you need additional styles and classes or if you need to edit the default styles and classes, you can do so in the styles editor.

Using Links

- Remember to copy the links that you need from the repository screen.
- Check the DB Interactions window for the screen. If you have unlinked table views, you need to either copy the link from a repository screen or create a new link.

  If the DB Interactions window lists invalid links because of missing table views, add the table views to the screen. If the table views are not needed, you can leave the invalid links on the screen or delete them.
- Check your link properties. You may need to change the link type or reverse the parent-child values.
- Do not specify circular links among two or more table views. If `link1` sets `tview1` as the parent table view and `tview2` as the child table view, then another link on the screen cannot have `tview2` as the parent and `tview1` as the child.

Setting Widget Properties

- To execute database queries based on specific column values, set the Use in Where and Operator properties in the Database category for the applicable widgets.
- A widget can only belong to one table view; however, two widgets can reference the same database column.

Errors in the Transaction Manager

The following transaction manager error messages are listed in alphabetical order with a possible cause and solution for each message. Those containing an error constant are stored in the JAM message file. Those without an error constant are caused by errors in SQL generation.

`Bad arguments (DM_BAD_ARGS)`

Cause: General error. One cause is that the `START` command was issued without a transaction name. Another cause is that a bad value was specified for the return code in a hook function.
Action: n/a

`Bad field name, #, or subscript at line` *line-number*

Cause: This standard JPL error generally indicates the JPL procedure or variable causing the error. One cause which is not a syntax error is using the property API to query for the value of server view, table view or link when it is not in the current traversal tree or for the value of `num_key_columns` when a database modification command is not in effect.
Action: Edit the JPL procedure.

`Bad mode (DM_TM_BAD_MODE)`

Cause: Command availability varies according to the transaction mode.
Action: 1) Refer to page 346 for the command availability in each mode. 2) Use the `COPY_FOR_UPDATE` and `COPY_FOR_VIEW` commands, which set the mode, when appropriate. 3) For menu items and push buttons, set the Class property which controls the active/inactive property according to the transaction mode.

`Column` *column-name* `not found in table view` *table-view-name* `specified in link` *link-name*

Cause: 1) Invalid column name specified in the link's Relations property. 2) Parent and Child entries for the Relations need to be reversed.
Action: Edit the Relations property to contain the column names which join the two table views named in the link. Check that the column names exist in the corresponding Parent and Child table views.

`Discard all changes? (DM_TM_DISCARD_ALL)`

Cause: A transaction manger command was executed without saving the changes made to the onscreen data.
Action: Choose `Yes` to discard all changes. Choose `No` to return to the screen so that the `SAVE` command can be executed.

`Discard latest changes? (DM_TM_DISCARD_LATEST)`

Cause: A transaction manger command was executed without saving the changes made to a portion of the onscreen data.
Action: Choose `Yes` to discard all changes. Choose `No` to return to the screen so that the `SAVE` command can be executed.

`Error executing JAM/DBi command (DM_TM_DBI_ERROR)`

Cause: An error occurred while executing a command in one of JAM's database drivers.
Action: Refer to error for action.

`Error in User Hook Function or Transaction Model (DM_TM_HOOK_MODEL_ERROR)`

Cause: This error lists whether a model or function is being accessed, the name of the model or function, and the event that failed. One common usage is to display the failed event after an error has been reported from the database engine.
Action: Generally, the engine error is more descriptive of the problem.

`Invalid field type for Version Column (DM_TM_VC_TYPE)`

Cause: Version columns must have the C Type property set to Int, Long, Float or Double.
Action: Change C Type property. As a result, database re-design may be necessary.

`Invalid sort order` *`type`* `specified in the sort–columns edit of tableview` *`table-view`*

Cause: Value entered for the sort type is invalid.
Action: Change the sort order type to `ASC` or `DESC`.

`Invalid widget` *`widget`* `specified in the sort–columns edit of tableview` *`table-view`*`.`

Cause: The Sort Widgets property does not contain a valid widget name.
Action: Check the table view's Sort Widgets property and make sure the widget is on the screen, remembering that it is the widget name and not the database column name that must be specified.

`Loop in transaction manager event processing (DM_TM_EVENT_LOOP)`

Cause: Transaction hook function specified in Function property is defined without passing it the event argument.
Action: Add `(event)` after the procedure name.

Cause: Transaction hook function has incorrect or invalid return code specified, for example, `TM_CHECK` instead of `TM_PROCEED` when no database driver statement was issued for that event.
Action: Change return code.

`Maximum depth exceeded`

Cause: 1) There is a circular link in the Parent and Child properties. 2) A link has both the Parent and Child properties set to the same table view.

Action: Check Parent and Child properties for each link, editing where necessary.

mode `does not permit command` *command* `(DM_TM_CMD_MODE)`

Cause: Command availability varies according to the transaction mode.

Action: 1) Refer to page 346 for the command availability in each mode. 2) Use the `COPY_FOR_UPDATE` and `COPY_FOR_VIEW` commands, which set the mode, when appropriate. 3) For menu items and push buttons, set the Class property which controls the active/inactive property according to the transaction mode.

`More than one row affected (DM_TM_ONE_ROW)`

Cause: `TM_CHECK_ONE_ROW`, which calls the `TM_TEST_ONE_ROW` event to check that `@dmrowcount` is equal to 1, has been set as the return code either in the transaction model or in a hook function.

Action: Change the return code in the model or hook function. Change the SQL generation, for example, by expanding the Primary Key values so that only one row is changed. Check the data in the database to make sure that duplicate key values have not been entered.

`No rows affected (DM_TM_SOME_ROWS)`

Cause: `TM_CHECK_SOME_ROWS`, which calls the `TM_TEST_SOME_ROWS` event to check that `@dmrowcount` is equal to or greater than 1, has been set as the return code either in the transaction model or in a hook function.

Action: If error is valid, do nothing. Otherwise, change the return code in the model or hook function.

`No select columns specified, first table view` *table-view*`.`

Cause: For all the members of this table view, either the Column Name property is blank or the Use In Select property is set to No.

Action: Set the appropriate properties for each widget.

`No such command as` *command* `(DM_TM_NO_SUCH_CMD)`

Cause: The syntax of `sm_tm_command` is incorrect.

Action: Edit the call to `sm_tm_command` so that a valid command name is the first parameter of its quoted command string.

`No such scope as in` *scope* `(DM_TM_NO_SUCH_SCOPE)`

Cause: The syntax of `sm_tm_command` is incorrect.

Action: Edit the call to `sm_tm_command` so that a valid scope parameter follows the command and table view parameters.

`No such table view as in` *table-view* `(DM_TM_NO_SUCH_TV)`

Cause: The syntax of `sm_tm_command` is incorrect.

Action: Edit the call to `sm_tm_command` so that a valid table view name follows the command parameter in the quoted command string.

`Primary key not specified for updatable Tableview` *table-view* `(DM_TM_PRIMARY KEY)`

Cause: For commands that could result in database modifications, like `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE`, the transaction manager checks that the primary key fields for a table view are available.
Action: 1) Specify the table view's primary keys in the Primary Keys property.

`Root table view name not supplied or not valid (DM_TM_NO_ROOT)`

Cause: Table view parameter supplied with `START` command is not valid.
Action: Edit `START` command specification.

Cause: More than one table view appears on a screen and there are no link widgets.
Action: Create a link widget with the appropriate Parent, Child and Relations property settings.

`Table name not specified for tableview` *table-view*`.`

Cause: Table property is blank for this table view.
Action: Enter the name of the database table in the table view's Table property. For the format needed by a specific database driver, refer to the Database Driver section.

`Table name not specified for Tableview (DM_TM_TBLNAME)`

Cause: Table property is blank for this table view.
Action: Enter the name of the database table in the table view's Table property. For the format needed by a specific database driver, refer to the Database Driver section.

`Tableview` *table-view* `is updatable but its primary key is incomplete.`

Cause: For commands that could result in database modifications, like `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE`, the primary key fields of an updatable table view must either be a member of that table view or one of its parent table views. It does not have to be in the direct parent, it can be in the "grandparent" table view.
Action: 1) Add the field to the desired table view. 2) Check the link's Relations property to see if the shared fields between the table views is complete. 3) Change the table view to non-updatable. 4) Check the link's Relations property to make sure the relation type is valid.

`Transaction model not found (DM_TM_NO_MODEL)`

Cause: Model specified in table view or screen properties is not initialized.
Action: Specify valid model or leave blank to use standard model.

`Transaction unspecified or unavailable (DM_TM_NO_TRANSACTION)`

Cause: Transaction manager command was called in an unnamed JPL procedure. Since JAM calls the unnamed JPL procedure before it calls the `START` command on screen entry, an error occurs.
Action: Call the command after the `START` command has been invoked, for example, in the screen entry procedure.

Cause: More than one table view appears on a screen and there are no link widgets.
Action: Create a link widget with the appropriate Parent, Child and Relations property settings.

Cause: `START` command was not issued because of error in table view tree or because the table view parameter specified with the command was invalid.
Action: Check Parent and Child properties. Check specification of additional `START` commands.

`Unable to synchronize server view (DM_TM_SYNCH_SV)`

Cause: For all updatable table views, the transaction manager synchronizes the table views when a command that could modify data is issued, for example, `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE`.
Action: 1) If possible, set all the members of the same table view whose Use in Update property is set to Yes to the same number of occurrences. 2) Change the Synchronization property to No. For more information , refer to page 377.

`User Hook Function not found (DM_TM_NO_HOOK)`

Cause: The table view's Function property specifies a name that is not available either as a JPL procedure or as a prototyped function.
Action: 1) Check the value of the Function property. 2) Check the prototyped function list. 3) Check that the JPL procedure name matches the Function property and that the JPL module is available.

`Version Column setting on widget is incompatible with the properties` *property_name*

Cause: If a widget's Version Column property is Yes, then the properties In Delete Where and In Update Where must be set to No.
Action: 1) Set the properties to the correct values.

SECTION SIX

Application Issues

Input/Output Processing

This chapter describes:

- How keyboard input is processed — including which library functions are called to carry out such processing.
- How JAM and JAM applications use the information encoded in the video file to process output.

All user input to a JAM application is processed through a keyboard translation file or table before being handled by JAM. Similarly, all output to the display monitor is processed through a video mapping table.

This translation of input and output is done to avoid code specific to particular displays or terminals, and thereby preserve terminal independence. JAM and JAM applications can run on a variety of terminals, provided that the appropriate keyboard and video configuration files are identified. These configuration files are used by the application at initialization to establish the keyboard and video translation.

Processing Keyboard Input

Keystrokes are processed in three steps:

1. The sequence of characters generated by one key is identified.
2. The sequence is translated to an internal value, or logical character.
3. The internal value is either acted upon or returned to the application (key routing).

All three steps, described in this section, are table-driven. Hooks are provided at several points for application processing; refer to page 115.

Logical Keys

JAM processes characters internally as logical values, which usually correspond to the physical ASCII codes used by terminal keyboards and displays. JAM uses the key translation file to map specific physical keys or sequences of physical keys to logical values, and the video file's `MODE` and `GRAPH` entries to map logical characters to video output. For most keys, such as displayable data characters, no explicit mapping is necessary. Certain ranges of logical characters are interpreted specially by JAM:

`0x0100` to `0x01ff`	Operations such as tab, scrolling, cursor motion
`0x6101` to `0x7801`	Function keys PF1 to PF24
`0x4101` to `0x5801`	Shifted function keys SPF1 to SPF24
`0x4103` to `0x5a03`	ALT keys ALTA to ALTZ
`0x6102` to `0x7802`	Application keys APP1 to APP24

Key Translation

The first two steps in JAM's processing of keyboard input—identification and translation—are controlled by the binary key translation file, loaded at initialization. JAM finds the file's name in a setup file or in the environment (refer to the *Configuration Guide* for details). The binary file is derived from an ASCII file that you can modify with any text editor.

JAM assumes that the first input character of a multi-character key sequence is a control character in the ASCII chart (0x00–0x1f, 0x7f, 0x80–0x9f, or 0xff) and

attempts to translate the character to a single logical key. Characters outside this range are assumed to be displayable characters and are not translated.

Note: *This algorithm assumes that a timing interval (*`KBD_DELAY` *entry in the video file) has not been specified. For more information, refer to page 111 in the Configuration Guide.*

On receiving a control character, the keyboard input function `sm_getkey` searches the key translation file for a sequence beginning with that character. If no match is found on the first character, JAM accepts the key without translation. If a match is found on the first character, an exact match, `sm_getkey` returns the indicated value. The search continues through subsequent characters until one of the following conditions is true:

- An exact match on *n* characters is found and the *n*th+1 character in the file is 0, or *n* is 6. In this case, the value in the file is returned.
- An exact match is found on *n*–1 characters but not on *n*. In this case, `sm_getkey` attempts to flush the sequence of characters returned by the key.

The latter condition is of some importance: if the user presses a function key that is not defined in the file, JAM must know where the key sequence ends. The following algorithm is then used:

- The file is searched for all entries that match the first *n*–1 characters and are of the same type in the *n*th character, where the types are digit, control character, letter, and punctuation. The smallest of the total lengths of these entries is assumed to be the length of the sequence produced by the key.
- If there is no entry matches by type at the *n*th character, the shortest sequence that matches on *n*–1 characters is used. Hence, `sm_getkey` can distinguish, for example, between the sequences `ESC O x`, `ESC [ A`, and `ESC [ 1 0 ~`.

with timing interval set

If you have a `KBD_DELAY` entry in your video file, you can specify key sequences in the key translation file that are substrings of other sequences. For example, the sequences `ESC` and `ESC [ C` can both have logical values, even though one is a substring of the other. In this case, JAM waits the specified timing interval, as indicated in the `KBD_DELAY` entry, between processing characters to determine if a character is a single keystroke or belongs to a sequence of keystrokes.

Key Routing

The third step in processing keyboard input is handled by the library function `sm_input`. This function calls `sm_getkey` to obtain the translated value of the key. It then decides what to do based on the following rules:

value greater than 0x1ff

If the logical value is greater than 0x1ff, `sm_input` returns the value as the return code.

value between 0x01 and 0x1ff

If the value is between 0x01 and 0x1ff, the key is translated via the key translation file. The processing of the key is then determined by a routing table. You can alter the default behavior of keys (cursor control) within this range with the library function `sm_keyoption` as well as set the routing information for a particular key. The routing value consists of two bits, examined independently, so four different actions are possible:

- If neither bit is set, the key is ignored.
- If the EXECUTE bit is set and the logical value is in the range 0x01 to 0xff, it is written to the screen (as interpreted by the `GRAPH` entry in the video file, if one exists). If the value is in the range 0x100 to 0x1ff, the appropriate action (Tab, field erase, etc.) is taken.
- If the RETURN bit is set, `sm_input` returns the logical value to the caller; otherwise, `sm_getkey` is called for another value.
- If both bits are set, the key is executed and then returned.

The default settings are ignored for ASCII and extended ASCII control characters (0x01 – 0x1f, 0x7f, 0x80 – 0x9f, 0xff), and EXECUTE only for all others. The default setting for displayable characters is EXECUTE. All other ASCII and extended ASCII characters are ignored. The function keys (PF1 to PF24, SPF1 to SPF24, APP1 to APP24, and ABORT) are not handled through the routing table. Their routing is always RETURN, and cannot be altered. All other function keys (EXIT, SPGU etc.) are initially set to EXECUTE.

changing key actions at runtime

You can program your application to change key actions at runtime by using `sm_keyoption`. For example, to disable the Backtab key, you execute the following function:

```
sm_keyoption(BACK, KEY_ROUTING, KEY_IGNORE)
```

To make the field erase key return to the application program, use

```
sm_keyoption(FERA, KEY_ROUTING, RETURN)
```

Logical key mnemonics are defined in the `smkeys.h` file in the include directory.

Processing Terminal Output

JAM defines a set of logical screen operations (such as positioning the cursor) and stores the character sequences for performing these operations in a video file specific to the display. Logical display operations and the coding of sequences are described in Chapter 7 of the *Configuration Guide*.

This section describes the ways in which JAM uses and ultimately, the way applications use the information encoded in the video files to determine how and what output your terminal displays.

How JAM Handles Output

JAM uses a *delayed write* output scheme to minimize unnecessary and redundant output to the display. No output at all is done until the display must be updated, either because keyboard input is solicited or the library function `sm_flush` is called. Instead, the runtime system does screen updates in memory and keeps track of the display positions. Flushing begins when the keyboard is opened; but if you type a character while flushing is in progress, the runtime system processes it before sending any more output to the display. Therefore, you can type ahead on slow displays. You can force the display to be updated by calling `sm_flush`.

Graphics Characters and Alternate Character Sets

Many terminals support the display of graphics or special characters through alternate character sets. JYACC provides 8-bit alternate character sets (for example, those that translate from IBM PC extended character to Latin-1). These tables can be installed by calling the library function `sm_xlate_table`. Control sequences switch the terminal among the various sets, and characters in the standard ASCII range are displayed differently in different sets. JAM supports 8-bit to 7-bit translations via the `MODEx` and `GRAPH` entries in the video file.

The seven `MODEx` sequences (where x is 0 to 6) switch the terminal into a particular character set. `MODE0` must be the normal character set. The `GRAPH` command maps logical characters to the mode and physical character necessary to display them. It consists of a number of entries whose form is

```
logical value = mode physical-character
```

When JAM needs to output `logical value` it first transmits the sequence that switches to `mode`, then transmits `physical-character`. It keeps track of the current mode to avoid redundant mode switches when a string of characters in one mode (such as a graphics border) is being written. `MODE4` through `MODE6` switch the mode for a single character only.

Writing Portable Applications

This chapter describes features of hardware and operating system software that can cause JAM 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.

Terminal Dependencies

Some of the general differences found from terminal to terminal are described in this section. In addition, recommendations and considerations are included to help ensure that your application can be ported to terminals that differ from your development environment.

display area

JAM can run on display terminals of any size. On character-based terminals without a separately addressable status line, JAM 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.

display attributes

Different terminals support different sets of attributes. JAM 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

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.

key labels

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 the `sm_keylabel` to return a printable name of the logical key.

Ensuring Portability with Library Functions

The header file `smmach.h` contains 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.
- Pathname and command line argument separator characters.
- Pointer alignment and structure padding.

The header file is thoroughly commented. Follow the directions in the file and use the information pertinent to your machine and operating system.

Writing International Applications

This chapter describes:

- Usage of JAM's 8-bit internationalization capabilities to support language-dependent applications.
- Translation concerns and issues.
- Translation of your JAM applications for international distribution.

JAM's Approach to Internationalization

JAM's support for 8-bit international character sets lets you customize JAM applications for use in non-English-speaking countries. This means that an application can be made to look native to the country in which it is being used. All prompts and messages can appear in the appropriate language, and customs for formatting dates, currency fields and the like can be observed. So while you might develop an application in English, you can make sure that your application can be easily adapted to a non-English-speaking audience as well as to a multilingual audience.

The capabilities described here are specific to those languages in which characters can be represented in 8 bits of information and those that use a left-to-right entry order. JAM does not support the complexities associated with languages that observe other conventions.

Supported Features

- JAM uses 8-bit character data without appropriating a bit for internal use.
- Date and time formats and currency formats are specified by you in the JAM message file, and therefore can be completely language-independent.
- JAM 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.
- JAM library functions `sm_is_yes` and `sm_query_msg` use the characters designated in the `SM_YES` and `SM_NO` entries in the JAM message file. Therefore, if you translate the message file, the screen use and display of those values are automatically internationalized. These functions use `toupper` to recognize upper-case variations.

Localization

JAM applications can be localized by taking the following steps:

1. Translate all screens in the application via the screen editor or by using `f2asc` (page 565 and translate the ASCII output to the desired languages. Refer to page 490 for other translation options and considerations.
2. Translate and recompile the JAM message file and any application message files.
3. Translate the documentation.

8-Bit Character Data

JAM 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 the use of graphics for borders, etc. Any unused entries in a character set, such as 0x01 through 0x1f or 0x80 through 0x9f, can be mapped to line graphics symbols.

JAM rarely, if ever, interprets characters present in screens or entered from the keyboard. Internally it merely manipulates numbers. Any meaning as an alphabetic character, graphics symbol, or whatever, is generally irrelevant to JAM. 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, there are some terminals that generate a sequence to represent an international character. If so, you can use a text editor to map the byte sequences into 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 473 on keyboard input and terminal output.

Date and Time Fields

The mnemonics for specifying date and time formats are stored in the JAM message file, and therefore are easily accessible. In addition, they are stored internally in a tokenized form. This provides two major benefits:

- No parsing of format is required at runtime, thus speeding up processing and reducing memory requirements.
- Date and time specifications are in formats that can be customized to local language, thus accommodating both English-speaking and non-English-speaking developers.

For example, an English-speaking programmer can create a date field with the format `mon/day/year`, and it might show up on a French system as `mois/jour/annee`. Interchanging of month and day is covered later in this chapter.

Table 51 shows the default message file entries for date and time mnemonics.

Table 51. Default message file entries for date and time substitution variables

Date/Time tag	Substitution variable	Token	Description
FM_YR4	YR4	%4y	4 digit year
FM_YR2	YR2	%2y	2 digit year
FM_MON	MON	%m	month number
FM_MON2	MON2	%0m	month number, zero fill
FM_DATE	DATE	%d	date (day of month)
FM_DATE	DATE2	%0d	date, zero fill
FM_HOUR	HR	%h	hour
FM_HOUR	HR2	%0h	hour, zero fill
FM_MIN	MIN	%M	minute
FM_MIN2	MIN2	%0M	minute, zero fill
FM_SEC	SEC	%s	seconds
FM_SEC2	SEC2	%0s	seconds, zero fill
FM_YRDA	YDAY	%+d	day of the year
FM_AMPM	AMPM	%p	am/pm
FM_DAYA	DAYA	%3d	abbreviated day name
FM_DAYL	DAYL	%*d	long day name
FM_MONA	MONA	%3m	abbreviated month name
FM_MONL	MONL	%*m	long month name

changing the date/time substitution variables

The date and time substitution variables correspond to ANSI standards. You can change them to suit your own needs by simply editing the JAM message file and then running `msg2bin` to make the changes available to JAM. For example, to change the substitution variable for a 4 digit year from `YR4` to `YYYY`, change this message file line:

```
FM_YR4 = YR4
```

to the following format:

```
FM_YR4 = YYYY
```

Next, run `msg2bin`. For information about this utility, refer to page 49 in the *Configuration Guide*.

If all development is done in one language, the fact that different mnemonics for date and time formats can be used for different languages is unimportant. It is

important, however, to modify an application to operate in a different language. Only the text on the screens and the message file should require changes.

Consider a screen with a date field of the form `DAYA MONA DATE, YR4`. If executed on a system with an English message file it might appear as follows:

```
Mon Apr 4, 1989
```

On a French system, it might look like this:

```
Lun Avr 4, 1989
```

translating names of months and days

This happens without changing the date format. Only the names and abbreviations of the months and days are changed. These are stored in the message file, so it is a simple matter to convert them.

editing the date format

Now consider a date field which in English appears as `mm/dd/yyyy`, but in French should appear as `dd-mm-yyyy`. In this case, the date format itself must be modified. For this reason, 10 additional formats are provided. For instance, you can specify a new date mnemonic in the message file; call it `REGULAR DATE`. In the English message file this can be equated to `mm/dd/yyyy` and in the French message file to `dd-mm-yyyy`. Thus, if the date format is specified as `REGULAR DATE`, only the message file, not the screens, needs to be changed to convert the date field to French format.

For this capability, both the mnemonics *and* what they represent are specified in the message file. The actual formats are stored in the message file in tokenized form so that there is no need for a parser.

Table 52 shows the default message file entries for the extra date mnemonics.

Table 52. Default date/time substitution variables and corresponding formats

Message file date/time tag	Substitution Variable	Token	Corresponding format tag	Default format
FM_0MN_DEF_DT	DEFAULT	%0f	SM_0DEF_DTIME	%m/%d/%2y %h:%0M
FM_1MN_DEF_DT	DEFAULT DATE	%1f	SM_1DEF_DTIME	%m/%d/%2y
FM_2MN_DEF_DT	DEFAULT TIME	%2f	SM_2DEF_DTIME	%h:%0M
*	DEFAULT	%3f	SM_3DEF_DTIME	%m/%d/%2y %h:%0M
*	DEFAULT	%4f	SM_4DEF_DTIME	%m/%d/%2y %h:%0M
*	DEFAULT	%5f	SM_5DEF_DTIME	%m/%d/%2y %h:%0M
*	DEFAULT	%6f	SM_6DEF_DTIME	%m/%d/%2y %h:%0M

*FM_3MN_DEF_DT through FM_9MN_DEFDT are undefined in the distributed JAM message file.

Message file date/time tag	Substitution Variable	Token	Corresponding format tag	Default format
*	DEFAULT	%7f	SM_7DEF_DTIME	%m/%d/%2y %h:%0M
*	DEFAULT	%8f	SM_8DEF_DTIME	%m/%d/%2y %h:%0M
*	DEFAULT	%9f	SM_9DEF_DTIME	%m/%d/%2y %h:%0M

*FM_3MN_DEF_DT through FM_9MN_DEFDT are undefined in the distributed JAM message file.

So, if you specify a date field with the format DEFAULT DATE, it appears in mm/dd/yy form. If you change this message file line:

```
SM_1DEF_DTIME = %m/%d/%2y
```

to this:

```
SM_1DEF_DTIME = %d-%m-%2y
```

the date appears in dd-mm-yy form. To change the substitution variable for this date format to REGULAR DATE, modify the message entry FM_1MN_DEF_DT as follows:

```
FM_1MN_DEF_DT = REGULAR DATE
```

Currency Fields

The formatting capabilities for currency fields support any convention you desire. As with date and time formats, a default format is supplied that causes the actual format to be taken from the JAM message file.

You can specify the following items for currency fields:

- Radix separator or decimal symbol — Usually period or comma.
- Minimum and maximum number of decimal places.
- Thousands separator — Usually period, comma or blank.
- Currency symbol — Up to 5 characters and its placement.

including spaces in currency fields

If you type a currency symbol into a regular field on a screen, you cannot include trailing spaces, because they are always stripped off. So, to specify a leading currency symbol separated from the data by a space, for example, FF 123.456,78, you must use the message file. For this reason, the period (.) can be used to signify a space when entered into the currency field. A period in the message file for this purpose appears as a period.

Currency formats are defined as *rmxtpccccc*:

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. for example, a comma or period; `b` for a blank.
p	Placement of currency symbol: `l` for left, `r` for right, or `m` indicating at decimal point.
ccccc	Currency symbol: up to 5 characters, including blank spaces.

Therefore, in the format `".22,l$"`, the period/decimal point represents the *r*; 2 represents *m* and indicates the minimum number of decimal places; 2 represents *x* and indicates the maximum number of decimal places; the comma represents *t* and serves as the thousands' separator; l represents *p* and indicates that the currency symbol be placed on the left; the dollar sign ($) represents *ccccc*, the currency symbol.

Therefore, if you specify a currency field with the format `Local currency`, it appears in `$999,999.99` form. If you change this message file entry:

```
SM_0DEF_CURR = ".22,l$"
```

to this format:

```
SM_0DEF_CURR = ",22.lFF"
```

the field displays data in the form `FF 999.99,99`.

changing a currency mnemonic

To change the mnemonic for this currency field, modify the message entry `FM_0MN_CURRDEF`. Table 53 shows the default formats and currency mnemonics for message file entries.

Table 53. Default message entries for defining currency formats

Currency tag	Substitution variable	Corresponding format tag	Default Format
`SM_0MN_CURRDEF`	`Local currency`	`SM_0DEF_CURR`	`".22,l$"`
`SM_1MN_CURRDEF`	`2 decimal places w/ commas`	`SM_1DEF_CURR`	`".22,"`
`SM_2MN_CURRDEF`	`0 decimal places w/ commas`	`SM_2DEF_CURR`	`".00,"`

*`SM_3MN_CURRDEF` through `SM_9MN_CURRDEF` are undefined in the JAM message file.

Currency tag	Substitution variable	Corresponding format tag	Default Format
SM_3MN_CURRDEF	DEFAULT	SM_3DEF_CURR	".22,"
through	DEFAULT	SM_4DEF_CURR	".22,"
SM_9MN_CURRDEF	DEFAULT	SM_5DEF_CURR	".22,"
are undefined in the	DEFAULT	SM_6DEF_CURR	".22,"
JAM message file	DEFAULT	SM_7DEF_CURR	".22,"
	DEFAULT	SM_8DEF_CURR	".22,"
	DEFAULT	SM_9DEF_CURR	".22,"

*SM_3MN_CURRDEF through SM_9MN_CURRDEF are undefined in the JAM message file.

Decimal Symbols

JAM accommodates three decimal symbols which are used in different circumstances:

System Decimal Symbol
The character obtained from the operating system (if supported by the operating system) (refer to the *Installation Notes* for your operating system), and used by C library routines like `atof` and `sprintf`. The default is period.

Local Decimal Symbol
Used when local customs are followed (dot in English; comma in French). Obtained from the operating system if supported by the operating system) (see the JAM *Installation Notes* for your operating system). The Local Decimal Symbol may be specified in the message file (tag entry `SM_DECIMAL`), in which case it overrides the system decimal symbol. Period is the default if no symbol is specified in the message file and if the operating system does not supply one.

Field Decimal Symbol
Specified for a given field if that field is not observing local conventions.

The sections below describe the circumstances under which each of the different symbols is used.

Keystroke Filter Translation

The one time that JAM requires some knowledge of the meaning of the data is while enforcing the keystroke, or character, filters on a field. The filters are digits only, numeric, alphabetic, alphanumeric, yes/no, edit mask, and regular expression.

To validate the data, JAM uses the standard C macros: `isdigit`, `isalpha`, etc. JAM assumes that the operating system supplies these macros in a form suitable for international use. In the absence of such operating system support, care should be taken when using these capabilities.

Special code is used to process numeric fields since C does not provide an "isnumeric" macro. If the field has a currency edit, JAM uses the Field Decimal Symbol to validate the numeric entry. If the field has no currency edit or the currency edit has no decimal symbol specified, JAM uses the Local Decimal Symbol.

Yes/no fields have always been internationalized in that the yes and no characters (`y` and `n` in English) are specified in the message file. Although some vendors supply information about these characters, the proposed ANSI standard does not address the issue. Therefore, for reasons of portability, JAM continues to use the message file for this data.

Upper and lower case fields will also behave properly provided that `toupper`and related functions are language dependent. The present code assumes that the return from `toupper` is appropriate for an upper case field. Therefore a lower case letter can appear in such a field if there is no upper case equivalent for that letter. (The German "double s" has no upper case equivalent.)

In processing regular expressions, JAM uses the ASCII collating sequence for 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.

Translation Considerations

This section describes some of the things you should take into account when you release your application to an international audience. You can easily modify JAM to meet your particular needs and in some cases, JAM handles the translation internally. This section describes:

- How JAM interprets language-specific information.
- Things to consider when you are in the process of developing an application that needs to be internationalized.

Translating Status and Error Messages

Runtime messages and prompts produced by JAM are stored in the JAM message file so they can be easily localized. Each message is a complete phrase or sentence.

Message components are structured in such a way to make translation to a non-English sentence structure can be easily managed.

In addition, you might consider adding an identifier to each message so that editing a translated message file can be easily handled. This can facilitate the maintenance of multiple message files as well.

Logical key mnemonics such as XMIT and EXIT are not translated to other languages, nor are the mnemonics used in the video file, so the internal processing of the utilities need not be modified.

Refer to page 47 in the *Configuration Guide* for details on modifying the JAM message file.

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. How you release these language-specific screens can be handled in a variety of ways as well, regardless of when the localization process takes places. You can use JAM configuration variables to identify the desired set of screens. For example:

using configuration variables

- Create multiple libraries; each one contains a set of screens translated to a specific language. By setting the `SMFLIBS` configuration variable either at distribution, installation, or runtime, you or a user can access the desired language-specific library.
- Store sets of screens in language-specific directories. By setting the `SMPATH` configuration variable you or a user can define which directory to search at runtime. For example, your application might have a Spanish directory as well as many other directories. Upon installation, the user can specify the desired directory path in the `SMPATH` variable.
- Create sets of screens corresponding to each language; name the screens using an extension that identifies the language, such as `logon.eng` for English and `logon.fre` for French. Use the `SMFEXTENSION` variable, which identifies screen filename extensions, to define which screens to open. To implement this change dynamically, you can use the library function `sm_soption`. In this way, the user can choose, for example, a radio button, which sets the variable `SMFEXTENSION` to open the `.fre` screens.

The following sections describe only some of the other methods you can consider when you are developing your 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 (refer to page 66) 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 (refer to page 565) 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.

Once the ASCII translations are made, they can be converted back to screens in binary format (with `f2asc`) and distributed accordingly.

advantages

Advantages to these methods are:

- 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.

disadvantages

Disadvantages to these methods are:

- Maintenance across 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

An installation translation means that the application is packaged with more than one language, and the desired language is installed. You can provide an installation mechanism whereby the user can set a JAM configuration variable (see page 491) to point to and open a set of language-specific screens.

Advantages to this method is that the end user can decide which language to install.

Disadvantages to this method are:

- Requires disk space to accommodate storage of multiple sets of screens.
- Languages cannot be changed dynamically at runtime.

Runtime Translation

A runtime translation means that the end user can dynamically change languages at runtime. Depending on your users requirements, they may only need to select a preferred language at start up, or they may 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. Essentially, you provide some mechanism whereby the user can choose which language to display. For example, on a logon screen you could

provide radio buttons that correspond to each supported language. The user can choose the desired language. This in turn, would set a configuration variable to point to and open the appropriate library, directory, or set of screens.

An advantage of this method is that a multilingual organization can easily run the application; each user can choose their preferred, or native language without requiring a reinstallation of the software.

A disadvantage of this method is that the installation requires enough disk space to accommodate all the 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.

Advantages to this method are:

- All translations exist in one place with each screen binary.
- The user can choose, while working in the application, which language to display.

A disadvantage to this method is that some translations require more space than others; your screens must be designed with these limitations in mind.

Interpreting Range Checks

of numeric data

Range checks for numeric data are handled by the C library routine `atof` (assuming that the “strip” routine works properly).

of alphabetic data

One of the major issues for internationalization is the collating sequence of a language. For dictionary or telephone book processing the problem is particularly troublesome. For example, upper- and lower-case letters are compared equally. Also, in a telephone book, `St.` and `Saint` are compared equally, hyphens are ignored, etc. In some languages even less demanding applications pose severe problems. 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 proposed ANSI standard specifies a routine, `strcoll`, that can be used to expand the word into a format suitable for comparison by `strcmp`. These routines assume that the data supplied is a word in the local language. They will give unexpected results on non-language data.

JAM is not designed to process languages in a way that requires such niceties. It does sort names of fields and other objects, but that is done only to speed look-up. As long as the sort routine and the search routine use the same algorithm, things will work.

of non-language data

In JAM, range checks are often given on non-language data. For example a menu selection might have a range of `a` to `d`. In certain languages an umlaut would fall into that range if a language-specific comparison was made. This effect would complicate screen design. Different screens would be needed for different countries, even if they used the same language.

The C routines `strcmp` and `memcmp` are used to range check on non-language data. The routines compare the internal values of the characters, without regard to their meanings in the local language.

Interpreting Math Calculations

The keywords `@sum` (yields the sum of all occurrences in given array) and `@date` (yields the number of days between 1/1/1753 and the argument to `@date`), though language-specific, can be computed accurately based on formats stored in the JAM message file. Computations with dates assume the Gregorian calendar. No provision is made for other calendars.

JAM Debugger

Whereas we may intuitively think of an application's behavior as being dictated by user actions, it is more accurate to say that an application's behavior is determined by the interactions of the application's components, *one* of which is user actions. The components of a JAM application are:

- JAM events, internal actions that cause a change in the processing flow of the application, such as a screen exit.
- JPL code.
- Data within JAM variables.
- JAM screens.
- Widgets, such as controls for your JAM screens.
- User actions, more generally, user events.

Debuggers are essential tools for programmers building complex applications. The JAM debugger is provided as an aid to JAM application developers. The JAM debugger analyzes the application components and allows you to view the application's execution from various perspectives with varying degrees of detail.

Debugger Features

The debugger lets you perform these tasks:

- Step through events, stopping at your discretion to examine data.

- Set breakpoints at events or JPL code on which to interrupt program execution.

  Breakpoints can be set on GUI events such as screen and field events, program execution events such as database and transaction manager events, as well as source code locations. A further refinement of event filtering is available for certain GUI events: breakpoints on screen, field, group, and grid events can be further restricted to specific sub-events.

- Examine application data:

  - data in JAM variables and arrays
  - screen properties
  - widget properties
  - monitor the value of a JPL variable or expression

- Step through your JPL code.

  An improved source browser screen allows you to view source code modules from disk files, open libraries, memory resident modules, public JPL modules, the window stack and the program stack. Breakpoints can be set on their contents.

- Call a function.

  When program execution is interrupted, you can direct the debugger to invoke an installed function or a JPL procedure, or evaluate a JPL expression.

- Stop at a change in a variable or expression.

- Run the debugger in expert mode.

  Set the debugger to expert mode and take advantage of advanced features from the Tools menubar option, such as: calling installed functions, application data watch, sorting breakpoints and other useful features. In normal mode (the default), breakpoint events are predefined, requiring only activation by the user to make use of event filtering.

- Review debugger activity in the log file.

  Debugger messages can be written to a user-specified log file. An option is available to dump the contents of all current debugger windows to the log file.

How the Debugger Works

Many debuggers do their work by invoking the applications they analyze. The JAM debugger is different. It is invoked by your application, that is, by JAM.

When the debugger is linked in and initialized, as it is in the development executable `jamdev`, JAM will notify it 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, giving no indication of its presence. Some events, however, cause it to emerge from the background—*awaken*—to halt the application and display its state.

The DBUG key immediately awakens the debugger. On other events, the debugger recalls what you have directed it to do and awakens accordingly—for instance, if the 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 actions: provide additional analysis of the application, modify data, set breakpoints, or put it back to sleep and return control to your application.

The View Menu: Debugger 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 View menu by choosing from these options:

Status

Displays the current debugger operation or the event currently being traced.

Source Code

The Source Code window displays JPL for internal or external modules. A breakpoint at the current line in the displayed code (where the text cursor is) can be set or un-set by choosing Breaks⇒Toggle Location Break. A breakpoint can be set or un-set at any line in the displayed code 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 505.

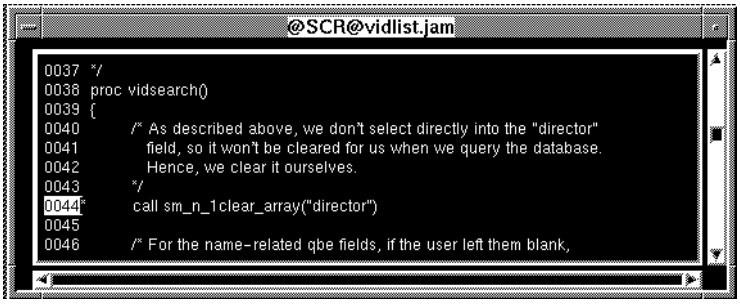


Figure 30. The Source Code window displays a file with a breakpoint set at line 44.

Breakpoints

This window lists all the breakpoints. Activation of a breakpoint is indicated by a "+" (active) or "–" (inactive) preceding the breakpoint. Activate or deactivate the breakpoint currently selected by choosing Breaks⇒Enable or Breaks⇒Disable. Alternatively, the activation state of a breakpoint can be toggled by double-clicking on the line. Use Breaks⇒Select All if you want to enable or disable all breakpoints listed.

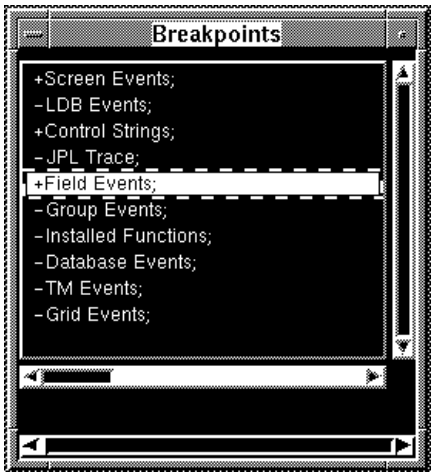


Figure 31. The Breakpoints window shows currently identified breakpoints.

In normal mode, the following predefined breakpoints are listed in the Breakpoints window: Screen Events, LDB Events, Control Strings, JPL Trace, Field Events, Group Events, Installed Functions, Database Events, TM Events and Grid Events. Any location breakpoints that are set will also appear in the window, as will any breakpoints you create in expert mode. Note that if the debugger is run in expert mode, the predefined list of breakpoints will be removed. Refer to page 512 for more information on setting breakpoints.

If a selected breakpoint is a location breakpoint, you can choose Breaks⇒Show Source to view the source code.

For more information on this window, refer to page 512.

Data Watch

In this window you can 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

During JPL or control string execution, this window shows the hierarchy of nested calls to procedures/control strings.

Figure 32. View the call hierarchy in the Event Stack window.

Pending Keys
Displays which keys are pushed onto the input queue.

Configuring the Debugger

Debugger operation preferences are set from the Options menu, and saved in the file `jamdebug.cfg`, in the current working directory. You should not manually modify this file.

Log File Preferences

Debugger activity can be logged to a user-specified file. Log file preferences are set in the Status Log Options window. Choose Options⇒Status Log... to raise the Status Log Options window.

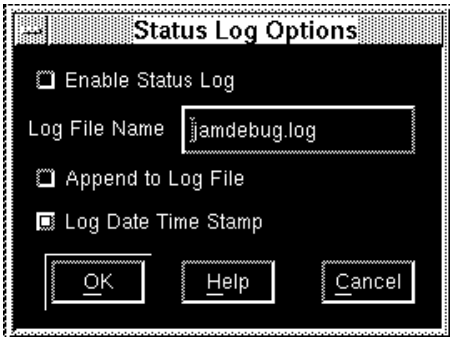


Figure 33. The Status Log Options dialog allows you to specify your log file preferences.

In this window you can:

- Enable the log file – all events will automatically be logged from the start of the session.

- Specify the log file to which debugger activity messages should be logged. The default file is `jamdebug.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. Note that performance may improve on some platforms if this option is not set.

While in the debugger you can view the contents of the log file at any time by choosing File⇒Open⇒Log File.

Other Debugger Preferences

The following debugger preferences are available directly from the Options menu :

Save Preferences on Exit

Save Preferences on Exit automatically saves for future sessions the window configuration and other debug settings that are in effect when you exit this session. This also has the effect of saving the contents of any Data Watch and Breakpoints settings. Contrast with File⇒Save Preferences, which saves preferences immediately.

Expert Mode

Choose whether to run the debugger in expert mode.

Auto Raise/Close

This option provides for automatic window management. When set, upon entry, 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

Set Animation to execute the debugger in animation mode. The debugger will show changes to its windows as the occur, while waiting for a breakpoint to be executed.

Trace Database Warnings

Include all warnings in Database events tracked by the debugger.

Trace TM Warnings

Include all warnings in transaction manager events tracked by the debugger.

Accessing the Debugger

JAM is shipped with the debugger already linked in. The debugger runs in the background during test and application modes. Once the debugger has been enabled, debugger windows automatically come to the foreground when a break event or breakpoint occurs. You can also manually break execution and open the debugger windows by pressing the DBUG key, or by choosing Options⇒Debugger. The debugger can be setup and activated in test and application mode, and it can be enabled in edit mode for immediate activation in test mode.

Enabling the Debugger from the Screen Editor

To enable the debugger from the screen editor, choose Options⇒Enable Debugger from the menu bar to instruct the debugger to awaken immediately upon entry into test mode.

Database Import Tracing in the Screen Editor

The debugger can be invoked from the screen editor to trace transaction manager import JPL. To trace any JPL import code, it must be in ASCII format. In the JAM distribution, the JDB import table code exists in binary format (`dmjdb.bin`). You must first convert any database import code to ASCII format with the `f2asc` utility. Follow these steps to trace TM import code in the screen editor:

1. Be sure the JPL import code is in ASCII format.
2. Connect to the database and open the repository.
3. Enter test mode.
4. Enter the debugger, choose View⇒Breakpoints and activate the TM Events breakpoint.
5. Exit to the screen editor.
6. Choose Options⇒Debugger Config... Check Enable Debugger and Trace TM Import.

7. Choose File⇒Import⇒Database Table to wake up the debugger and display the code in a window.

8. You can now use the function keys to trace through the JPL code:

Key	Function
F8	Trace⇒To Event⇒This Level
F9	Trace⇒To Event⇒Any Level
F10	Trace⇒To Breakpoint

Accessing the Debugger in Test or Application Mode

Within test mode, you can enter the debugger by choosing Options⇒Debugger, or by pressing the DBUG key. To put the debugger to sleep and return to test mode, choose File⇒Resume Application.

From application mode, if you have not yet opened your application, or if the application menu bar is not displayed, the debugger can be entered by choosing Options⇒Debugger from the JAM menu bar, or by pressing the DBUG key. If the application menu bar is displayed, you can enter the debugger by pressing the DBUG key. Alternatively, you can switch menu bar scope to `jamdev` by pressing the SFTS logical key, from where you will have access to Options⇒Debugger.

Exiting the Debugger

The File menu contains two options that let you exit a debugger session:

- File⇒Resume Application puts the debugger to sleep. JAM preserves the previous debugger state; if the debugger is re-entered, all break points and configuration preferences remain unchanged. However, all breakpoints and break events are ignored until you explicitly reactivated the debugger. If you invoked the debugger from test mode, Resume Application will return you to test mode.

- File⇒Exit Application aborts the current JPL execution and quits the debugger. If you entered the debugger from application mode, Exit Application gives you the option of returning to application mode or exiting JAM to the operating system. If you were in test mode, it returns you to the screen editor. Use this option to break out of infinite loops.

The Debugger Menu Bar

The debugger menu bar provides you with 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.

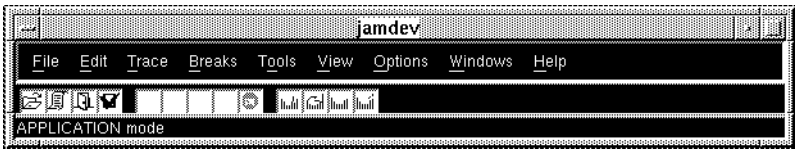


Figure 34. The Debugger menu bar. Note that the Tools menu only appears if the Debugger is run in expert mode.

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. Whichever is chosen, the data will be displayed in the Source Code window. For more information on Source Code window operations, refer to page 505.

Close Window
Choose to close the active window. If you attempt to close the last remaining window, the debugger presents a message dialog requesting confirmation that you want to quit the debugger.

Save Preferences
Choose to save for future debugger sessions the window configuration and other debug settings that are in effect at the present time. Using this setting, you can establish the preferences you want for future debugger sessions, but remain free to make any changes in the present session that will not affect the saved preferences. Contrast with Options⇒Save Preferences on Exit, which will save them at the state they are in when you exit. For more information on setting debugger configurations, refer to page 499.

Resume Application
Choose to put the debugger to sleep. Exit the debugger with Resume Application when you wish to continue executing your application.

Exit Application
Choose to exit the debugger when you wish to quit execution of your application and return to application mode, or quit JAM entirely.

For more information on Resume Application and Exit Application, refer to page 502.

Tools

The Tools menu can be only accessed when the debugger is running in expert mode, accomplished via the Options menu. The following features are available from the Tools menu:

Application Data
Choose to instantly view and access any variable displayed in the Source Code window. For information on using the Application Data window, refer to page 518.

Call...
Choose Tools⇒Call... to raise the Call Installed Function window. Enter the name of any available prototyped function, along with any arguments. The return value, if any, will be displayed in the status window after the function returns. The function call will be logged to the log file if it has been enabled.

Sort Breakpoints
This feature sorts all breakpoints listed in the Breakpoints window. First the activated breakpoints are listed alphabetically, then the inactive breakpoints.

Sort Watch Data
This Tools feature allows you to sort all variables and/or expressions listed in the Data Watch window. For further information on this window, refer to page 517.

Write Windows to Log
Use this option to write the contents of the debugger screens to the log file. The log file has to be enabled for this to take affect. The log file is enabled by setting the Enable Status Log check box found on the Status Log Options window; this window is opened from the Options menu.

Once the log file is enabled, this Tools feature writes the contents of the active debugger screens to the log file.

View

The debugger provides several windows to monitor your application’s execution. Refer to page 497 for a full description of the options available from the View menu.

Windows Menu

Options available from the Windows menu allow you to:

- Bring a chosen window into focus.
- Arrange the debugger windows in either tiled or cascade format.
- Choose an application window to examine its programmatic components. For information on viewing application screen information, refer to page 509.

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 505.

Trace

The Trace features allow you to step through program execution one event or breakpoint at a time. For more information on tracing, refer to page 511.

Breaks

Commands from the Breaks menu are used to establish and manipulate break points in your application—places where execution will be interrupted and the debugger will assume control. For further information on setting breakpoints, refer to page 512.

Options

Choose from the Options menu to set your debugger preferences. For information on debugger preferences and configuration, refer to page 499.

Viewing JPL

Choose View⇒Source Code to open the Source Code window and view source code from your application. Use this window to view screen, widget, and grid

validation JPL, as well as external JPL modules. Breakpoints can be set on any line of code that is displayed in the window. The Edit menu provides string search capabilities that you can use to locate strings in the displayed code.

The Source Code window can display any JPL code contained within your application. The JPL code can be:

- *Current* source code — the source that is currently being executed. The 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 may 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 to gain access to the debugger's file browsing mechanism. 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.

File Browsing: Begin at the Open Source Module Window

The Open Source Module window is used to read a module into the Source Code window. It is also used to select a module from which to establish a location breakpoint.

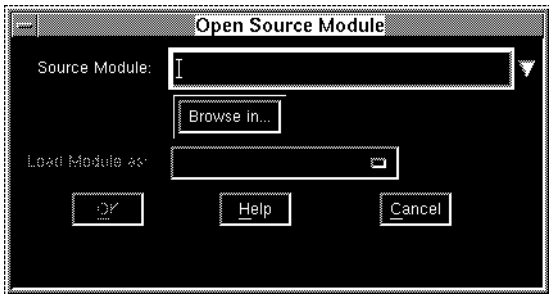


Figure 35. The Open Source Module is used to read a module into the Source Code window or select a source file for the Edit Breakpoint window in Location mode.

To read a JPL module into the Source Code window:

9. Choose File⇒Open⇒Source Module.

   The debugger opens the Open Source Module window.

10. Choose `Browse in` and select the location of the module from the menu selections that appear:

    - Disk Files — any screen or JPL file
    - Open Libraries — JPL or screens in any open library
    - 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

    If Disk Files is chosen, a file selection box opens from which you can choose the file. If any of the other locations are chosen, the Browser window is opened. The title bar on the browser window will reflect the location chosen. If there is no source code available for the chosen location, a message dialog informs you.

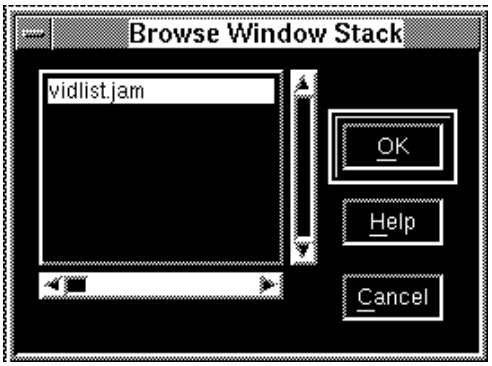


Figure 36. The Browse window displays the available JPL code corresponding to the browser location selection. Here, Browse in, Window Stack is chosen, and this is reflected in the title bar.

11. Select the file from the file selection box or the browser. Choose OK to accept the selection and dismiss the file selection box or the browse dialog and send the module to the Open Source Module window. The type of the module selected is automatically set in the `Load Module as:` field. The type will be one of:

- JPL File
- Screen JPL
- Field JPL
- Grid JPL
- Text File

If a request to open a JAM screen file (`.jam`) contains JPL modules of more than one type, for instance screen, field, and grid widget JPL, the choices are presented in the `Load Module as:` field, from which you must select the type of the JPL module you are interested in.

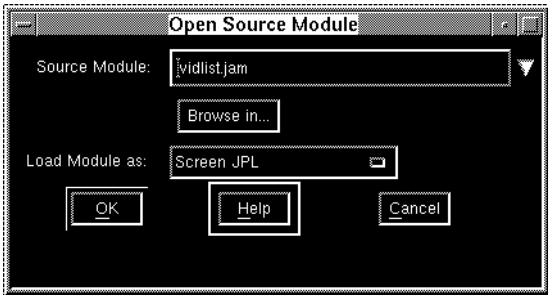


Figure 37. A JAM screen file on disk is selected for viewing.

12. Choose OK in the Open Source Module window to accept the choice and dismiss the window.

The debugger displays the Source Code window with the contents of the specified module.

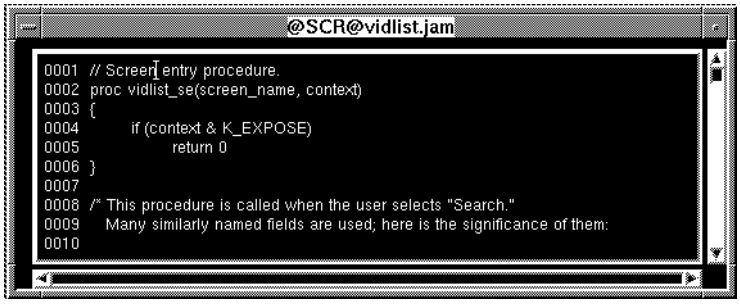


Figure 38. The Source Code window displays screen JPL code from the video list screen from the Videobiz sample application.

Set Breakpoints in the Source Window

To toggle a breakpoint on or off on the current line of code displayed in the Source Code window, choose Breaks⇒Toggle Location Break, or simply double-click anywhere in the line. Breakpoints are identified by an asterisk following the line number.

Viewing Application Screen Information

The debugger provides access to all the programmatic components of your application screens. Select the screen to examine by choosing Windows⇒Application⇒*application-screen*. The following choices are available from Application Window on the menu bar:

Screen JPL

Select this option to view screen JPL in the Source Code window. This is an alternative to using File⇒Open⇒Source Module (refer to page 505).

Screen Information

This selection opens the Screen Inquire window that displays screen information:

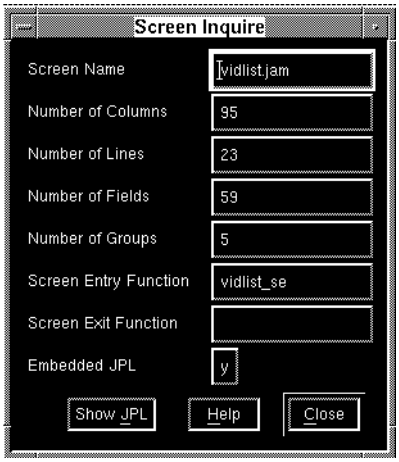


Figure 39. This Screen Inquire window displays useful information on a screen from the Videobiz sample application.

The Show JPL button is enabled if the screen has its own JPL module.

Field Information

This selection opens the Field Inquire window that displays the information related to fields on your application screen:

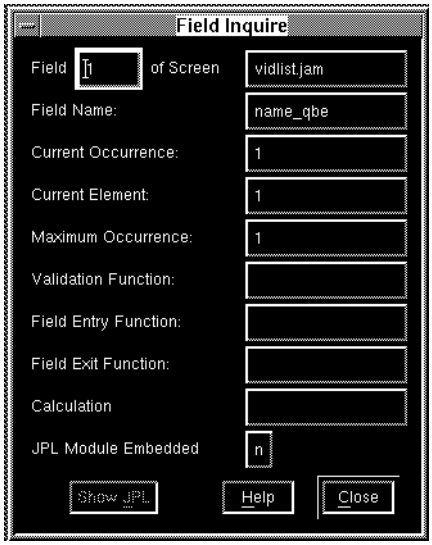


Figure 40. The Field Inquire window displays useful information on a field on a screen from the Videobiz sample application.

If the field has its own JPL, the Show JPL button is enabled.

Group Information

This selection opens the Group Inquire window that displays information related to groups on the screen:

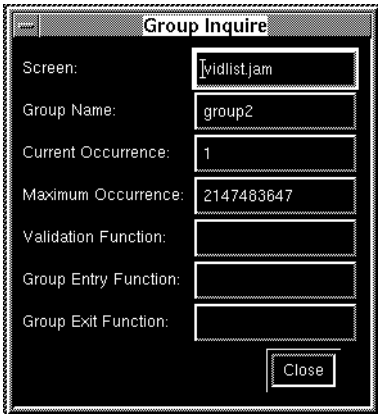


Figure 41. The Group Information window.

Control Strings

This option invokes the Control Strings window, which shows all function keys and JAM logical keys that are assigned control strings at the application and the screen level.

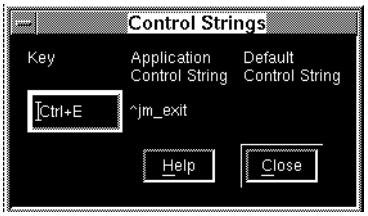


Figure 42. The Control Strings window shows that the exit JAM function is attached to the JAM EXIT key.

Done

Choose Done to return to the debugger menu bar.

Note: *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 518 for information on using the Application Data window. Refer to page 517 for information on using the Data Watch window. For information on JPL properties syntax, refer to page 24 in the Language Reference.*

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. In normal (non-expert) mode, the Trace menu has these options:

Trace⇒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.

Trace⇒To Breakpoint

Program execution resumes until the next breakpoint is reached.

Expert Mode

If you run the debugger in expert mode you can fine tune tracing to differentiate between parent, child, and same-level events. Trace⇒To Event⇒Any Level is identical to normal mode Trace⇒To Any Event, and Trace⇒To Breakpoint is the same in both modes.

Trace⇒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.

Trace⇒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.

Trace⇒To Event⇒Higher Level
The debugger recognizes 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.

Note that the debugger provides toolbar icons and keyboard accelerators to execute tracing commands, providing a more efficient method to step through program execution.

Automatic Stepping Using Animation

The debugger can be set to run on ‘automatic pilot’, where application execution events can be observed hands-off in the various View windows. To enable this feature, toggle on Animation (choose Options⇒Animation), and then 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 it is explicitly turned off from the Options menu. You can also interrupt animation with the EXIT key.

Setting Breakpoints

The debugger recognizes all major execution events in a JAM application as potential breakpoints—that is, events on which the debugger assumes control.

Breakpoints can be set at:

- Locations in code: JPL statements (within screen, widget, and grid validation, as well as in files).
- Program execution events and sub-events.
- A change in the value of a variable or expression.

Note: *The debugger gets control at every execution event and potential breakpoint. The debugger then decides whether or not to awaken, and whether or not to break execution. It makes the decision whether or not to awaken based on the user's Trace choice, and whether breakpoint conditions are satisfied.*

If you are not running the debugger in expert mode, you can set location breakpoints in the Source Code window and modify the predefined execution events in the Breakpoints window. However, in expert mode, you also have available the Edit Breakpoint window, from which you can add and modify breakpoints of both types.

Location Breakpoints

Location breakpoints are JPL statements that are marked in order to stop program execution and bring the debugger to the foreground whenever JAM encounters it. You can set breakpoints on any JPL code that is displayed in the Source Code window. To set a breakpoint, select a line of code and choose Breaks⇒Toggle Location Break, or double-click on the line. The debugger prefixes the line with an asterisk (*).

To clear a breakpoint from a specific JPL statement, use the same procedure for setting the breakpoint—setting and clearing breakpoints is a toggle-state process.

Breakpoints can be set on both called-up source code as well as current (active) source code. Refer to page 505 for further information.

Setting Breakpoints on Execution Events

The JAM debugger provides breakpoint access 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.

Normal Mode

Choose View⇒Breakpoints to display the Breakpoints window. The predefined events are already listed. To activate an event selected in the list, either double-click on the item or choose Breaks⇒Enable. Disable a breakpoint by double-click-

ing or choosing Breaks⇒Disable. Activation of an event breakpoint is indicated by a plus sign (+) prefix. If the breakpoint is not activated, it is prefixed with a minus (–).

You can set breakpoints on these predefined execution events:

Screen Events
Break on all screen entry and exit events. Screen events have sub-events defined, which you may selectively enable in expert mode.

LDB Events
Break on each LDB read or write operation.

Control Strings
Break on execution of each control string. If you have embedded control strings, the debugger breaks on each of these depending on the current “step” level.

JPL Trace
Break on every line of JPL execution. To step through each line of code, choose Trace⇒Breakpoint. To break at a specific line of line, toggle break on the line in the Source Code window and turn off JPL Trace in the Breakpoints window. Trace to Breakpoint will then continue execution and break at the specific line of code.

Field Events
Break on all field entry, validation, and exit events. Field events have sub-events defined, which can be accessed in expert mode.

Group Events
Break on all group entry and exit events. Group events have sub-events defined.

Installed Functions
Break when JAM is about to call any C function. Installed functions are C functions that are either installed by calling `sm_install` or are identified in `funclist.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, for instance in a control string, with the JPL `CALL` statement, or upon field entry.

Database Events
Break on any database interface event, including `DBMS` commands and SQL statements.

TM Events

Break on any transaction manager event. This includes transaction manager commands, requests and slices.

Grid Events

Break on any grid widget events. Grid events have sub-events defined, which can be accessed in expert mode.

Event Filtering in Expert Mode with the Edit Breakpoints Window

Expert mode provides access to sub-event filtering. In expert mode, the Breakpoints window is initially empty. Use the Edit Breakpoint window to add or modify breakpoints. The Edit Breakpoint screen is raised by choosing Breaks⇒Add when you want to add a breakpoint and/or by Breaks⇒Edit when you wish to edit an existing breakpoint.

The Edit Breakpoints window can be used to edit both location and event breakpoints, and to perform other actions. The window has two modes: Event and Location. Select the appropriate check box: Break At Event for event (the default setting), and Break At Location to edit source code breakpoints.

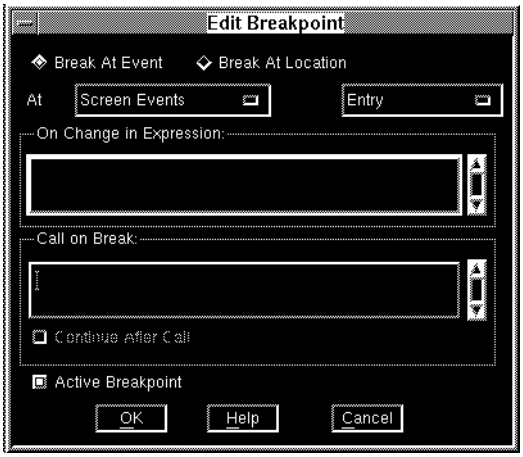


Figure 43. In expert mode, use the Edit Breakpoints window to add or modify breakpoints of all types.

Break on Change in Expression

You can direct the debugger to make activation of a breakpoint dependent upon the value of an expression. If a valid JPL expression is entered in the `On Change in Expression` text widget, the debugger will only stop at the event/location breakpoint (assuming it has been activated) if the value of the expression changes,

i.e., the value changes from what it was when Trace⇒To Breakpoint was chosen. Set break at `Any Event` if you want the debugger to awaken as soon as the change occurs.

Call on Break

You can direct the debugger to call a specified function each time the breakpoint is reached. Enter the JPL or installed function to be executed in the `Call on Break` text widget. You also have the option of instructing the debugger to call the function without stopping execution; check `Continue After Call` if this behavior is desired.

Event Mode

Select from the pop-up menu of events next to `At`. The sub-event menu appears when the event selected has sub-events defined. These events and their sub-events are listed in the following table:

Table 54. Events and their corresponding sub-events for which breakpoints can be applied.

Screen Events	Field Events	Group Events	Grid Events
Any sub-event	Any sub-event	Any sub-event	Any sub-event
Entry	Entry	Entry	Entry
Exit	Exit	Exit	Exit
	Validation	Validation	Row entry
	Calculation		Row exit
			Validation

When you have selected the event breakpoint, choose OK to add it to the list in the Breakpoints window. If the Active Breakpoint box is checked, it will be added to the list activated (+). Uncheck this box if you wish to add it to the list in the inactive state (–).

Location Mode

The Edit Breakpoint window opens in Location mode when you are editing a location breakpoint. Typically, you will do this after double-clicking on a line in the Source Code window, which adds the location to the breakpoint window, from where you can select it.

If the Edit Breakpoint window is in Event mode initially, you can check `Break At Location` and the Edit Breakpoint window allows you to select a module and line number at which to establish a breakpoint. If the module and line number that you wish to break on are known to you, simply enter them in the `Module:` and `Line:` text widgets. Otherwise, you can choose `Browse...` to initiate file browsing. The Open Source Module window is opened.

For instruction on how to specify a file from the Open Source Module, refer to page 506. When you have chosen the module from Open Source Module, it is sent to the Edit Breakpoint window.

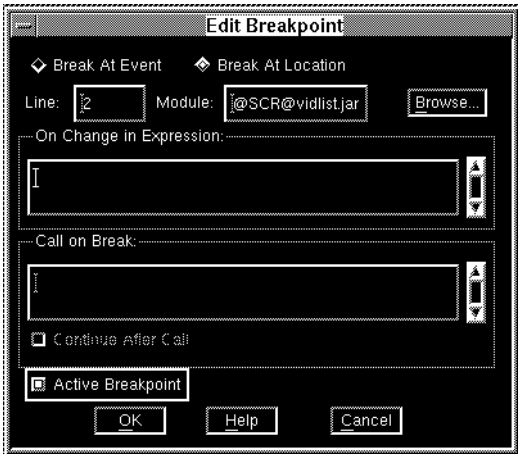


Figure 44. Use the Edit Breakpoint window in Location mode to set a code location breakpoint.

Add the Breakpoint

To add the location breakpoint, choose OK from the Edit Breakpoint window. The Edit Breakpoint window is dismissed and the location breakpoint is added to the list of breakpoints, visible in the Breakpoints window. If Active Breakpoint was checked on the Edit Breakpoint window, the breakpoint is automatically active; otherwise it is inactive. To find a particular line number, view the file in the Source Code window.

In general, you will find it easier to set location breakpoints by reading the code into the Source Code window and directly setting the breakpoint there. Refer to page 508 for further information on setting breakpoints in the Source Code window.

Monitoring Variables and JPL Expressions

When running the debugger you can examine the contents of variables or expressions and observe runtime changes as follows:

1. Open the Data Watch window by choosing View⇒Data Watch.
2. Type in the names of the variables or the expressions you want to observe.

   The debugger displays the values of the variables/expressions on the Data Watch window, updating them as execution proceeds. The location of any variables specified is also identified.

You can use Data Watch to inspect a property of the application, a screen, or widget, with a JPL expression. For more information on referencing JAM objects and properties, refer to page 24 in the *Language Reference*.

To clear a variable/expression, delete it from the list. To clear all variables/expressions, press `CLR`.

Modifying and Monitoring Application Data in Expert Mode

If you are running the debugger in expert mode, you can take advantage of the enhanced data watching abilities available with the Application Data window.

You can instantly view and access any variable displayed in the Source Code window by moving the cursor into the variable and then choosing Tools⇒Application Data to open the Application Data window.

The Application Data window provides the following capabilities:

- Displays the current value of the variable. If the variable is an array, you can choose which occurrences to view.
- Set a breakpoint at the touch of a button. The breakpoint will occur whenever the value of the variable or expression changes.
- Add the variable or expression to the Data Watch window.
- Change the current value of the variable.

Preparing Applications for Release

Before delivering an application to its users, you need to package it for delivery. There are several ways to package a JAM application for distribution. Typically, the screens, menus, and JPL are shipped in one or more libraries. The configuration files and the application executable are shipped as separate files. If desired, the screens, menus, JPL modules, and the configuration files can be shipped as part of the application executable.

This chapter outlines optimization options and requirements that precede delivery. You should also review the configuration guide for information on specific files and directories that JAM requires for distribution.

Required Files

The list of files that you must include in a distribution varies, depending on the application's components and the platform on which it runs.

Common Files

All JAM applications have the following files in common:

- JAM executable.
- JAM screens.

- Library of JAM screens `jam.lib`.
- Message file `msgfile.bin`.
- Video and keyboard files for all terminal types on which the application is to run.

GUI Files

GUI applications require resource files: on the Macintosh, the JAM preferences file inside the Preferences folder, `JAM7.INI` for Windows, and `XJam` for Motif.

Optional Files

The following files are optional, depending on your application's components and how it is configured:

- Library of JAM menu bars `jammn.lib`
- JPL files, either in binary or text format.
- Menu binary files.
- Default LDB screen `ldb.jam`.
- Message file in binary format for your own messages.
- Default setup file `smvars.bin`. You can specify the required setup and configuration variables elsewhere—for example, in your system's initialization file.

Specifying Files and Directories

You can use the `smvars` file to specify all the files and directories that JAM needs to run. The user only needs to set the `SMTERM` and `SMVARS` environment variables, which tell JAM which video and key files to use and where to find the application's files. Refer to page 5 in the *Configuration Guide* for more information about setting up your system environment.

Alternatively, you can specify required files through calls to JAM library 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 JAM binaries.
- `sm_msgread` reads a message file into memory.
- `sm_l_open` opens a JAM library.

Detailed information on each of these functions and their variants is available in the *Language Reference*.

Modifying Source Code

You can edit the source file `jmain.c` to change the default behavior of JAM applications. `jmain.c` has four functions defined in it: `main`, `initialize`, `start_up`, and `cleanup` 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`, `start_up`, and `clean_up`. Code necessary to your application can be inserted into the `main` routine. Any code inserted before `initialize` is executed before any JAM function has been executed.
- `initialize` allocates internal data structures and sets the terminal characteristics. Code inserted after `initialize` but before `start_up` is executed after JAM allocates internal data structures and sets the terminal characteristics, but before there are any screens and before there is a local data block.
- `start_up` creates local data blocks (LDBs) and the application's startup screen. Code inserted after `start_up` but before `clean_up` is executed after JAM exits the last screen, but before memory structures are deallocated and the terminal is reset.
- `cleanup` exits back to the operating system and restores the terminal's display state. Code called after `clean_up` is executed after all JAM functions have been executed.

If a finer granularity is needed, you can edit `initialize`, `start_up`, and `clean_up` themselves. Do so only if you understand JAM thoroughly.

Note: *In general, you should not modify* `jmain.c` *to install hook functions. Most hook functions can be declared in* `funclist.c`*. A few hook function types, however, must be installed before or during initialization in* `jmain.c`*. For more information on hook function installation,* *refer to page 115.*

Subsystem Installation

After the definition of the `main` function, there are a number of JAM subsystem macro definitions. They are all set to 0 by default. To turn on a subsystem, set the corresponding macro to 1. The following sections describe JAM subsystems.

ALT_SCROLLING
If the application installs and uses a custom scroll driver (described on page 529), this subsystem must be enabled.

MEMORY_SCREENS
If this subsystem is installed, screens displayed by the JAM can linked into the application as data and maintained in memory. If not installed, screens can only be read from disk. Installing this subsystem increases memory requirements but improves execution speed.

JTERM_COMPRESSION
This subsystem increases the communication efficiency and execution speed for applications when they are accessed by the terminal emulator J*term*. It increases the application's memory requirements.

Storing Screen and JPL Files in Libraries

JAM binaries—screens, JPL files, and menu bars—can all be stored and distributed in libraries. The `formlib` utility acts as the application librarian. For more information on this utility, refer to page 561.

Memory-Resident Screens

The screen editor creates binary screen files that are disk-resident. Memory-resident screens are much quicker to display than disk-resident screens, because no disk access is necessary to obtain the screen data. However, the screens must first be converted to source language modules with `bin2c` or a related utility, then compiled and linked with the application program. For information about converting screens with `bin2c`, refer to page 563.

You can install screens on a memory-resident list. This is a list maintained by JAM that specifies memory-resident screens and other binaries. When JAM attempts to open a screen, it first looks in this list for the requested screen; any screen found there is displayed from memory, while screens not in the list are sought on disk. 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.

The screen list is a pointer to an array of structures:

```
struct form_list
{
   char *form_name;
   char *form_ptr;
} *sm_memforms;
```

To initialize it, an application uses code as in the following example:

```
#include ”smdefs.h”
extern char mainform[], popup1[];
extern char popup2[], helpwin[];

struct form_list mrforms[] =
{
   ”mainform.jam”, mainform,
   ”popup1.jam”,   popup1,
   ”popup2.jam”,   popup2,
   ”helpwin.jam”,  helpwin,
   ””,             (char *)0
};
...
sm_formlist(mrforms);
```

The last entry in the screen list has an empty string for the name and a null pointer for the screen data. This marks the end of the list and is required. The call to `sm_formlist` adds the screens in the `form_list` structure to JAM’s memory-resident list.

JAM appends the extension found in the setup variable `SMFEXTENSION` to screen names that do not already contain an extension; take this into account when creating the screen list. JAM might also convert the name to uppercase before searching the screen list, as determined by the `SMFCASE` 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.

Note: *Because the JAM screen editor can only operate on disk files, altering memory-resident screens during program development requires a tedious cycle of test—edit—reinsert with `bin2c`—recompile. Therefore you should make screens memory-resident only at the very end of your development process.*

Memory-Resident Configuration Files

Any or all of the three configuration files required by JAM can be made memory-resident. You must create a C source file from the binary version of the file with `bin2c`. The source files created are not readily decipherable. The following code makes all three files memory-resident:

```
/* Memory-resident message, key, and
 * video files */
extern char key_file[];
extern char video_file[];
extern char msg_file[];

/* ...more declarations... */

sm_keyinit (key_file);
sm_vinit (video_file);
sm_msgread (”SM”, SM_MSGS, MSG_MEMORY|MSG_INIT, msg_file);
sm_jinitcrt (””);

/* ...possibly initialize function and
 * form lists   */

/* ...application code */
```

If a file is memory-resident, the corresponding environment variable or `SMVARS` entry is unnecessary.

Message File Options

If you need to conserve memory and have a large number of messages in message files, call `sm_msgread` with an argument of `MSG_DSK`. This avoids loading the message files into memory; instead, they are left open, and the messages are fetched from disk when needed. This uses up additional file descriptors, and buffering the open file consumes a certain amount of system memory; no significant benefit is gained unless your message files are very large.

Memory-Resident JPL

JPL public and memory-resident modules can be made memory-resident. First, compile the module with `jpl2bin` and convert the binary to a source language character array with `bin2c` or a related utility. Then, add the modules to the memory-resident list via `sm_formlist` and compile and link the source with your application.

Public modules can reside in files or libraries. Refer to page 7 in the *Language Reference* for an explanation of the various JPL modules. Making a JPL module memory-resident reduces I/O time and makes it part of the JAM executable. This can prevent accidental loss or editing of your JPL code by the end user.

JPL Versus C

Because JPL is an interpreted language, code execution undergoes an extra layer of interpretation that C avoids. In most cases, the total runtime difference is insignificant unless a JPL function is long or loops many times. In that case, it may pay to rewrite the procedure in C.

Minimizing Screen Output

Several entries in the JAM video file are not logically necessary, but are there solely to decrease the number of characters transmitted to paint a given screen. This can have a great impact on the response time of applications, especially on time-shared systems with low data rates; but it is noticeable even at 9600 baud.

For example, JAM can do all its cursor positioning with the CUP (absolute cursor position) command. However, it uses relative cursor position commands (CUU, CUD, CUF, CUB) if they are defined; they always require fewer characters to do the same job. Similarly, if the terminal can save and restore the cursor position itself (SCP, RCP), JAM uses those sequences instead of the more verbose CUP.

The global variable `I_NODISP` vsm also be used to decrease screen output. While this variable is set to 0 (via `sm_iset`), calls into the JAM library cause the internal screen image to be updated, but nothing is written to the actual display; the display can be brought up to date by resetting `I_NODISP` to 1 and calling `sm_rescreen`. With delayed write, this technique rarely necessary.

Small and Medium Memory Models

JAM applications assume usage of a large memory model. If you are compiling a DOS application that uses JAM library routines *and* you want it to use a small or medium memory model, change the declaration for `sm_fmalloc`—found in `smproto.h`—so that the return is explicitly cast to `char *` inststead of `VOIDPTR`. Otherwise, the application is liable to abort when it attempts to dereference a two-byte address for this function’s return value as if it were a four-byte pointer.

Stub Functions

Some JAM facilities can be omitted from an application if they are not used by defining certain literals in the application. This can result in substantial memory

savings; however, it requires that JAM libraries not be pre-linked or pre-bound—that is, not supported on all systems. You can stub out these facilities:

Table 55. Subsystems that you can stub out from an application

Subsystem	#define
Math package	NOCALC
Scrolling functions	NOSCROLL
Time and date functions	NOTIMEDATE
JAM help screens	NOHELP
Shifting fields	NOSHIFT
Range checking functions	NORANGE
Word wrap	NOWRAP
Field zoom expansion	NOZOOM
Regular expressions	NOREGEXP
Form libraries	NOFORMLIB
JPL	NOJPL
Runtime JPL compiler	NOJPLCOMP
Save/restore screen data	NOSRD
Local print	NOLPR
Area attributes	NOAREA
Window selection	NOWSEL
Keytop translation	NOLKEYLAB
Shift/scroll indicators	NOINDICATORS
User window resizing	NOWINSIZE
Mouse handling routines	NOMOUSE
Save screen to memory	NOSVSCREEN

To omit any one or combination of the above, #define the appropriate macro in your application, then #include the stubs file. This must only be done once, preferably in the application's main routine source file. For example, if the application does not use scrolling fields, you can omit the scrolling functions in the application source as follows:

```
#define NOSCROLL
#include ″smdefs.h″
#include ″smstubs.c″

main ()
{
   /* ...the application code... */
}
```

The effect of defining the macro and including `smstubs.c` is to declare stub routines in the application; this causes the linker not to add the real routines from JAM library to the application. The primary benefit lies in the amount of code space that is saved. The stubbing technique does not work on systems where the library is itself a linked entity, such as a sharable library.

If range, math, and JPL support are all stubbed out, you can also omit linking the C math library (`-lm` flag on UNIX systems, math library on MS-DOS systems).

If the runtime JPL compiler is stubbed out, file, public modules can still be used if they are precompiled with `jpl2bin`.

Alternative Scrolling

By default, storage of scrolling arrays is handled internally by JAM, which stores them in its own memory buffers. It is also possible for this data to be stored by the application, external to JAM—for example, in memory or disk. In this case, the application must install a *scrolling driver* which is called by JAM with an interface defined by JAM. Installation of a scrolling driver replaces JAM's default scroll driver. The driver is called to initialize the array, get and put occurrences, and so on.

An alternative scroll driver can reduce application memory usage when used to control the scrolling of large arrays. Scroll drivers can be freely mixed on a screen. Each driver can be specified to manage any number of arrays and any number of drivers can be used at once.

Normally, a non-scrolling widget can hold as many occurrences of data as the display allows. In character mode, this corresponds to the number of lines the widget occupies onscreen. The data held in non-scrolling widgets is kept as part of the normal screen data structure. However, by setting the Scrolling property to Yes, you can enter a number of occurrences that is equal to or greater than the number of visible occurrences. You can also enter 0 to specify an unlimited number of occurrences.

Because the amount of data kept in scrolling widgets can grow large, the data for offscreen occurrences is managed separately from onscreen. Management of offscreen data is handled either by JAM's default scroll driver, or by a custom-written driver, which you can specify in the Alt Scroll Func property. If this property is left blank or the name is invalid, JAM uses its own scroll driver.

JAM ships with the sources to two scrolling drivers in the `samples/altscroll` directory, and the header file which defines the interface is in `smaltsc.h`. JYACC customers are encouraged to develop their own scrolling drivers to suit their needs.

Array Geometry Properties

JAM defines an array and its size through its number of elements, occurrences, and its largest used occurrence ID. For more information on arrays, refer to page 79. The following terms are used in this chapter to describe an array's properties:

occurrences

The actual data held by an array, whether on- or offscreen.

element

An onscreen occurrence. An element refers to a fixed location on the screen, regardless of the actual data in it.

max_occurs

The maximum size of a scrolling array, expressed in number of occurrences, set by the array's Number of Occurrences property. The array's maximum can be unlimited if this property is set to 0. Consequently, the scrolling driver usually does not try to allocate buffers to hold `max_occurs` occurrences of data .

luid

JAM needs to know how many buffers are allocated for a scrolling array. This number is known as the largest used item (occurrence) ID or *luid*. This represents how many buffers are allocated, rather than how much data is in the array. The `luid` value grows as more data is put into the array, but shrinks only at explicit command.

num_occurs

The actual number of occurrences, equal to the occurrence number of the last non-blank entry in the array. The value `of num_occurs` can be expressed as follows: `num_occurs` <= `luid` <= `max_occurs`.

Installation

You can bundle multiple scrolling drivers into a JAM application. Scrolling drivers are installed in `funclist.c` in `sm_do_uinstalls`. Two types installations are important:

- A default scrolling driver, used when an array does not specify which driver to use or the specified driver is not installed.
- Installation of alternate drivers that are available in the executable.

The definition and installation of the default driver in `funclist.c` looks like this:

```
static struct fnc_data udfunc[] =
{
   SM_OLDFNC(”virtmem”, sm_vmbscroll),
};

static int udcount =
            sizeof (udfunc) / sizeof (struct fnc_data);
sm_install (DFLT_SCROLL_FUNC, udfunc, &udcount);
```

The definition and installation of the list of drivers in `funclist.c` looks like:

```
static struct fnc_data ufuncs[] =
{
   SM_OLDFNC(”virtmem”, sm_vmbscroll),
   SM_OLDFNC(”dosmem”, sm_mbscroll),
   SM_OLDFNC(”dummy”, adummy),
};

static int ucount =
            sizeof (ufuncs) / sizeof (struct fnc_data);
sm_install (SCROLL_FUNC, ufuncs, &ucount);
```

If no default scrolling driver is installed, JAM uses the one in `mb_scroll.c`, the memory-based scrolling driver.

JAM Interaction With Scrolling Drivers

When JAM initializes an application, it calls all scrolling drivers and asks them to install themselves. Similarly, when the application exits, JAM calls the driver for clean up. While the driver is active, JAM calls the routine once for each new scrolling array that JAM creates—for example, on screen open—telling it to initialize the array.

While the array is active, JAM moves data back and forth between itself and the driver and informs the driver of other changes to the array—for example, insertion or deletion of occurrences.

When the array is destroyed—for example, its window closes—the driver is called to release all the data associated with the array.

Scroll Driver Interface

All scroll drivers have a single entry point. JAM passes the driver two parameters. The first is a pointer to an `altsc_t` structure; the second parameter is an operation code, indicating what action JAM needs the scrolling driver to perform. Following is an example definition of a scrolling driver:

```
int
scroll_driver (as_ptr, action_code)
altsc_t *as_ptr;
int  action_code;
```

Scroll Driver Action Codes

The interface to the scrolling driver is through the mechanism of various action codes. Each action code represents one particular functionality. As can be seen by the table below, three codes are not currently used.

Table 56. Scroll driver action codes

Action	Explanation
`AS_DLT_FUNC`	Delete a range of lines.
`AS_GDATA_FUNC`	Get data from the driver.
`AS_GTSPC_FUNC`	Reserve space for occurrences.
`AS_INIT_FUNC`	Initialization, called when a new scrolling array is created—for example, a screen is open with a scrolling array using this driver.
`AS_INSRT_FUNC`	Insert blank lines.
`AS_INST_FUNC`	Installation: driver is being installed, called only once.
`AS_PDATA_FUNC`	Put data to the driver.
`AS_RESET_FUNC`	Reset, called when JAM is shutting down.
`AS_RLS_FUNC`	Release, called when a scrolling array is deleted—for example, the screen is being closed.

The altsc_t Structure

The following table describes the `altsc_t` structure, passed as the first parameter to a scrolling driver.

Table 57. The altsc_t structure

Member name	Description	Type
Driver-maintained information:		
`scrolldata`	Pointer to structure maintained by driver.	VOIDPTR
Scrolling information:*		
`luid`	ID of the largest-used occurrence	unsigned int
`max_items`	Maximum number of occurrence	unsigned int
`shs`	Largest non-blank occurrence	unsigned int
Occurrence information:		
`item`	occurrence number	unsigned int
`len`	Length of the occurrence	unsigned char
`attr`	Occurrence attributes	attrib_t
`valids[AS_VAL]`	Occurrence validation bits	unsigned char
`*text`	Occurrence data	unsigned char
Other parameters:		
`number`	Integer parameter	int

*This information is supplied on all calls except for `INIT` and `RESET`.

A pointer to an `altsc_t` structure is the first parameter to a scroll driver. The `altsc_t` structure serves two purposes:

- Lets the scrolling driver save information about an array between calls.
- Acts as a vehicle for passing data between the driver and JAM.

Because a scrolling driver can manage several arrays at once, it needs a way to keep track of the data for each array. The `scrolldata` member of the `altsc_t` structure serves this purpose. When the driver is called to `AS_INIT_FUNC`, the driver should store a pointer in the `scrolldata` member, usually to an internal structure. JAM assumes that the driver uses the pointer in `scrolldata` to access

the offscreen data. The `scrolldata` pointer can only be changed during the `INIT` call; all other calls pass back the same pointer saved from the `INIT` call.

The `altsc_t` structure also passes data into and out of the driver. The scrolling information parameters—`luid`, `max_items`, and `shs`—are passed so the driver knows the size of the array, current and potential. However, the `luid` and `shs` parameters are only completely accurate for the `AS_INSRT_FUNC` and `AS_DEL_FUNC` calls; otherwise, they are approximate. The occurrence information parameters—`item`, `len`, `attr`, `valids`, and `text`—are used to pass information about specific occurrences into and out of the driver. The other parameters —`number`, `vptr`—are used only for specific calls.

These members are discussed later in the description of each action.

Return Values

The scrolling driver generally returns 0 if the function is supported and it succeeds, -1 if the function is not supported, and some other non-zero value if the function fails. Exceptions are noted in the action code descriptions.

Action Codes

Many actions are required to update members of the `altsc_t` structure that is passed to the driver. Although the scrolling driver has one entry point, almost all drivers are implemented as a giant switch statement that calls other routines. For example, JAM's default scrolling driver `mbscroll.c` looks like this:

```
int
sm_mbscroll (as_ptr, option)
altsc_t *as_ptr;
int  option;
{
   int  retcode = 1;
   switch (option)
   {
   case AS_INIT_FUNC:
       retcode = mb_initscr (as_ptr);
       break;

   case AS_GDATA_FUNC:
       retcode = mb_getitem (as_ptr);
       break;

   case AS_PDATA_FUNC:
```

```
        retcode = mb_putitem (as_ptr);
        break;

    case AS_INSRT_FUNC:
        retcode = mb_insitem (as_ptr);
        break;

    case AS_DLT_FUNC:
        retcode = mb_delitem (as_ptr);
        break;

    case AS_GTSPC_FUNC:
        retcode = mb_setluid (as_ptr);
        break;

    case AS_RLS_FUNC:
        retcode = mb_rlsscrl (as_ptr);
        break;
    }
    return retcode;
}
```

The following sections describe the calling protocols and functionality expected for the routines associated with these action codes.

AS_INST_FUNC

Return Value

1 Success
0 Failure—the driver is not installed.

Description

`AS_INST_FUNC` expects a routine that is called once when JAM starts up. Note that if an application causes JAM to initialize and reset more than once, the function is called repeatedly. No buffer is passed to the driver—instead, it gets `NULL`.

This routine usually initializes the driver's global properties, allocate buffers, initialize virtual memory, open files. However, it can perform no tasks at all, especially if the initialization is performed for each array by `AS_INIT_FUNC`

AS_RESET_FUNC

Description

`AS_RESET_FUNC` expects a routine to call when JAM terminates with `sm_resetcrt`. The routine can perform any desired final clean up—deleting temporary files, freeing memory allocated in the `INST` function, and so on.

This routine has no input or output parameters and returns no value. It should only be called by JAM when it has released all active arrays with `AS_RLS_FUNC`.

AS_INIT_FUNC

Input Parameters

max_items

The maximum number of occurrences in the array. If the value of `max_items` subsequently changes, JAM calls the `AS_GTSPC_FUNC` function to inform the driver of the new limits.

len

Maximum width of the text of each occurrence.

Output Parameters

scrolldata

Set to point to a driver-specific data structure that keeps track of the data for the array. This value is associated with the array; all future calls to the scrolling driver that refer to this array get the `scrolldata` value passed in the `altsc_t` structure.

Return Value

0 Success
–1 Failure

Description

`AS_INIT_FUNC` expects a routine that is called when JAM starts handling scrolling data for an array. From the `max_items` and `len` parameters, the driver determines roughly how much data the array can hold.

AS_RLS_FUNC

Input Parameters

scrolldata
The pointer to the driver's internal buffer.

Return Value

0 Success
–1 Failure

Description

`AS_RLS_FUNC` expects a routine that is called when JAM destroys an array. The routine should then free any resources allocated for the array.

JAM ignores the return value from this call. Thus, the scrolling driver must inform the user about any failure and take the appropriate action—for example, exit the program.

AS_GDATA_FUNC

Input Parameters

scrolldata
The pointer to the driver's internal buffer.

item
The occurrence number to get.

len
Length of the occurrence. This represents the size of JAM's buffer; the scrolling driver should not overrun this length.

text
A buffer to get the text.

Output Parameters

attr
The display attribute of the occurrence.

text
The routine should fill the buffer pointed to by text with the text of the occurrence as passed to the routine by `PDATA`.

valids
The occurrence's validation bits; these are packed into a unique form.

Return Value

0 Success
–1 Failure

Description

`AS_GDATA_FUNC` expects a routine that gets the data of an occurrence in a scrolling array. This routine might be called because the occurrence scrolled onscreen, or in response to a request for data from a C or JPL routine—for example, a call by `sm_getfield`.

Occurrence data contains text, display attributes, and validation flags. JAM allocates a buffer to hold this data and passes a pointer to the buffer in the text member. The `AS_GDATA_FUNC` routine should update the `attr`, `text`, and `valids` members in the structure. These should be updated with values previously passed in by the `AS_PDATA_FUNC` routine, although a clever driver can manufacture them.

Note: *An empty occurrence must be blank-filled.*

AS_PDATA_FUNC

Input Parameters

scrolldata
The pointer to the driver's internal buffer.

item
The occurrence number to save.

len
Length of the occurrence.

attr
The display attribute of the occurrence.

valids
The validation bits of the occurrence.

text
The text to save.

Return Value

0 Success
–1 Failure

Description

`AS_PDATA_FUNC` expects a routine that is called when JAM wants to update offscreen data in the driver. The routine should save the contents of the `attr`, `valids`, and `text` members.

Note: *The routine cannot assume that text is null-terminated; rather, it should save all `len` bytes of it.*

AS_DLT_FUNC, AS_INSRT_FUNC

Input Parameters

scrolldata
The pointer to the driver's internal buffer.

item
The occurrence number to start at—the first deleted occurrence or first newly inserted occurrence.

number
The number of occurrences to delete or insert.

len
Length of the occurrence.

attr
The display attribute to give to new occurrences.

valids
The validation bits to give to new occurrences.

text
The text to put into new occurrences.

Return Value

Number of lines actually inserted/deleted, should equal `number`.

Description

`AS_DLT_FUNC` and `AS_INSRT_FUNC` expect routines to insert and delete array occurrences, respectively. These routines typically manipulate indexes or lists to implement the deletion/insertion.

New entries that are inserted into the array, or trailing occurrences that are left blank when occurrences are deleted, should have their `attr`, `valids`, and `text` set from the values that are passed in the structure. A `NULL` pointer for text indicates an empty string.

AS_GTSPC_FUNC

Input Parameters

scrolldata
The pointer to the driver's internal buffer.

number
The new `luid` to use.

Return Value

The number of occurrences it has resources allocated for. Usually, the return is the same as `number`.

Description

`AS_GTSPC_FUNC` expects a routine that tells the driver the largest occurrence, or `luid`, that it must keep track of. Any buffers or resources currently allocated for keeping track of data above `number` occurrences can be freed.

Dynamic Data Exchange

JAM supports Windows Dynamic Data Exchange (DDE), which lets applications share data through client/server links. JAM supports both client and server links with other applications. As a DDE server, a JAM application exports data from a field. As a DDE client, it imports data from another application into a field.

JAM as a Server

As a DDE server, JAM can export data from a named field to a client application. The client application specifies a DDE service, topic and item. In JAM, 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 JAM field. The request must include the name of the JAM server name, the screen name, and the field name. If the request succeeds, a DDE connection is created between that spreadsheet window and the JAM 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 JAM application, two conditions must be true:

- The JAM application must be running.
- The application must be enabled as a DDE server.

To enable JAM as a server, call `sm_dde_server_on` or include this setting in the initialization file `JAM7.INI`:

```
DDEServer=on
```

Creating Links

Clients can create links to a JAM 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 JAM application:

1. Copy data from a JAM 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 JAM screen.

Links Specified in Client Syntax

You can also create links to a JAM screen through explicit requests in the client application's DDE syntax. Refer to your client application's documentation for details on its DDE syntax.

Note: *Prefix the topic name with ampersands (*`&` *or* `&&`*) if you want the JAM screen to open as a stacked or sibling window.*

The following examples show DDE syntax for several widely used Windows applications:

Quattro Pro for Windows

```
@DDELINK([myJamApp|&&mainScreen.jam]"totalData")
```

MS Word for Windows

```
{DDEAUTO myJamApp &mainScreen.jam totalData \t}
```

MS Excel for Windows

```
=myJamApp|mainScreen.jam!totalData
```

Processing Links

It is the client application's responsibility to connect with a JAM 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 JAM application, it processes it as follows:

1. Finds a match between the specified service and a JAM application that is already running.
2. Checks whether the JAM 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 JAM application exits. JAM maintains links for a screen that is inactive or closed, and resumes data updates when the screen reopens. The client's connection to a JAM screen remains active until all links to the current screen are destroyed.

JAM destroys links only at the request of the client or when the JAM 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 JAM fields:

Hot Links
JAM 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
JAM 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
JAM 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, JAM supplies all occurrences in the array. Occurrences are separated by carriage returns (`\r`) and newlines (`\n`). Leading blanks in right-justified fields and trailing blanks in left-justified fields are omitted.

Data Conversion

JAM 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 JAM as a Server

You can disable JAM as a server at runtime through `sm_dd_server_off`. JAM continues to maintain all previous links to clients; however, it ignores all client requests that are made after this call.

JAM as a Client

When a JAM application acts as a DDE client, it imports data into fields from an external server application. JAM can request hot, warm, and cold links.

DDE can maintain multiple connections between the JAM application and server application; only one connection is allowed between a JAM screen and a given server window. Each connection can maintain multiple links.

For example, a JAM 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 JAM 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 JAM and automatically are written to its linked field.

Enabling Connections

Before JAM can connect to a server application, it must be enabled as a DDE client. To enable JAM as a client, call `sm_dde_client_on` or include this setting in the initialization file `JAM7.INI`:

```
DDEClient=on
```

Creating Links

As a DDE client, JAM can request hot, warm, and cold links. For more on link types, see page 550. You can create links in three ways:

- Call one of JAM's paste link functions, which get server data from the clipboard.
- Specify server data with one of JAM's connect functions.
- Specify server data in the initialization file.

Paste Links

You can paste server data from the Windows clipboard into a JAM field on the active screen with one of these JAM 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. JAM 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

JAM 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 JAM 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. Table 58 shows the syntax used by three widely used Windows applications.

Table 58. 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 JAM client and an Excel spreadsheet:

```
retval = sm_dde_client_connect_cold \
         ("Excel","C:\XL\SALES.XLS","R1C2","total")
```

Refer to the *Language Reference* for more information about these functions.

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 JAM contains a `JAM 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 58 shows the syntax used by three widely used Windows applications.

Processing Link Requests

A link request from a JAM 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 JAM 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 JAM 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—JAM posts an error message.

Updating Data from the Server

JAM 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 JAM that new data is available from the server. JAM 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` in the *Language Reference*.

DDE does not notify the JAM client of any changes in cold link data.

Note: *Because `sm_dde_request` can be called only for fields on the active screen, an application that uses warm links should queue notices for data on inactive screens.*

Array Data

JAM tries to update all occurrences in the array with server data. Data flows into the array starting with the first occurrence. When JAM reaches the end of the occurrence or encounters a tab, carriage return, or newline in the server data, it skips to the next occurrence. JAM 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 Server

When a screen closes, JAM destroys all links on that screen. You can also explicitly destroy links on the active screen with `sm_dde_client_disconnect`.

Disconnecting from a Server

JAM 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, JAM breaks the connection.

Execute Transactions

The execute transaction lets a client execute a command on a server. As a client, JAM can initiate execute transactions on a server application; as a server, JAM can be the recipient of commands issued by a client.

As a DDE client, JAM 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, JAM can receive a command issued by a client. For example, a Quattro Pro spreadsheet might contain this EXECUTE statement:

```
{EXECUTE B1, "^updateData.jpl"}
```

JAM 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, JAM can initiate poke transactions on a server application; as a server, JAM can be the recipient of commands issued by a client.

As a DDE client, JAM 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, JAM 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.

Mouse Interface

This chapter shows how to evaluate and process mouse events, mouse data, and contextual information. Topics include:

- Trapping mouse events.
- Using JAM 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 JAM'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 JAM's key change hook function, you can intercept single and double mouse clicks throughout the program. JAM'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 JAM 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 555. For more information on key change functions, refer to page 136.

```
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

JAM 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

Mouse Click Location

The library function `sm_ms_inquire` lets you test the last mouse click's line and column location on a JAM screen or on the physical display. Several runtime properties also offer access to the field and screen on which the last mouse click occurred.

Line and Column

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 JAM 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 );
   }
}
```

Field

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 `@jam` modifier. For example, this all-purpose code obtains the data in the last clicked-on occurrence of any field:

```
vars data
data = @widget(@jam–>mouse_field_name)[@jam–>mouse_field_occ]
```

Screen

`mouse_form` 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 `@jam` modifier as in this example:

```
vars mouse_screen
mouse_screen = @jam–>mouse_form
```

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 data for the middle button, if any. The three highest-order bits contain right button data.

Each bit within a three-bit segment can be set as follows, from lowest- to highest-order bit:

0/1 Up/down
1 Just pressed
1 Just released

For example, the bit settings returned for a just-initiated point and click operation—left button is down and just pressed—can be represented as follows:

Right Button			Middle Button			Left Button		
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:

Right Button			Middle Button			Left Button		
0	0	1	0	0	0	0	0	0

Only four combinations of bit settings are meaningful to JAM 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;
}
```

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:

1 Shift key is down
1 Ctrl key is down
1 Alt key is down

For example, a return value of 2 (`0 1 0`) indicates that the Ctrl key is down, while a return value of 5 (`1 0 1`) indicates that the Alt and Shift keys are both down. The second of these returns can be represented as follows:

Alt	Ctrl	Shift
1	0	1

In the following example, the return value of `sm_ms_inquire(MOUSE_FIELD)` is bitwise AND'd 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;
   }
```

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.

Note: *Ordinarily, you can use the key change function to intercept double mouse click events. For more information, refer to page 553.*

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.

GUI cursors

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.

character-mode behavior

Because character-mode JAM 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 JAM is awaiting input.

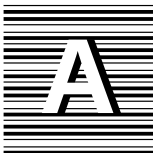
`SM_BUSY_CURSOR`
Changes the mouse pointer into the delay cursor.

`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. JAM restores the mouse pointer to the default cursor after JAM refreshes the screen.



Development Utilities

This appendix describes utilities that are useful in the development process.

Creating and Maintaining Libraries

JAM's `formlib` utility lets you create and maintain libraries of JAM screens, menus, and binary JPL files, as well as repositories.

Synopsis

```
formlib –c [ fluv ] library filename...]
formlib –d [ luv ] library filename...]
formlib –r [ luv ] library filename...]
formlib –t [ luv ] library filename...]
formlib –x [ fluv ] library filename...]
formlib –m library
formlib –g cfg-str library
formlib –{ s+|s– } [ –v ] library
```

Options and Arguments

–c

Create a new library that contains the files named.

–d
Delete the named files from the library.

–r
Replace/add the named files to the library.

–t
Generate a list of the library's contents.

–x
Extract the named files from the library. If none are named, all are extracted. Do not include wildcard specifiers in the file name.

–f
Allow the output file to overwrite an existing file.

–l
Convert binary file names to lower case before processing.

–u
Convert binary file names to upper case before processing.

–v
Generate list of files processed.

–m
Compact library/repository to eliminate wasted space.

–g
Define a configuration management string.

–s
Set (+) or clear (–) the SYSLIB flag.

Description

`formlib` creates libraries in which you can store JAM screens and binary JPL files. You can also store ASCII files in a JAM 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 existing libraries; for example, you can put a library 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 on a MS-DOS system puts all files with the `.jam` extension into the library `screenlib`:

```
formlib –c screenlib *.jam
```

The `–l` and `–u` options are useful for operating systems like UNIX that are case-sensitive. For example, this UNIX command creates the library `newlib` and adds all `*.jam` files in the current directory in it; all receive lowercase names—for example, `MAIN.JAM` is identified as `main.jam`.

```
formlib –cl newlib *
```

For information about source control management options, refer to page 361 in the *Editors Guide*.

Converting Binary Files to C Data Structures

`bin2c` converts JAM binaries—screens, menus, and JPL modules—into C character arrays.

Synopsis

`bin2c` *`[–fluv] ascii–file screen...`*

Options and Arguments

ascii–file
The name of the output file.

input-file
The name of the input file

–f
Overwrite an existing output file.

–l
Convert file names sent to output to lower case.

–u
Make array of `UCHAR` instead of `char`.

–v
Generate list of files processed.

Description

When `bin2c` creates the ASCII C file, it generate an array for each of the binary input files. An array in the file has one of these two form:

`char` *src–file*`[] = {` *contents of file* `};`

`UCHAR` *src–file*`[] = {` *contents of file* `};`

where *src–file* is the name of the source binary file with its path and extension stripped off. If you use the `-l` option, *src-file* is in 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 522. The following files can be made memory-resident:

- Screens
- JPL modules
- Menus
- Key translation files
- Setup variable files
- Video configuration files
- Message files

You cannot convert a file to its original binary form after using `bin2c`. JAM provides other utilities that permit two-way conversions between binary and ASCII formats. For screens, these utilities are `bin2hex` and `f2asc`.

Converting Binary Screens to Hex ASCII

`jamdev` creates binary screen files. Use `bin2hex` to convert these to and from hexadecimal for porting screens across different systems.

Synopsis

`bin2hex -c` *[* `flv` *] ascii–file screen...*

`bin2hex -x` *[* `flv` *] ascii–file*

Options and Arguments

ascii–file
With `-c`, *ascii–file* is the output file. All of the binary input files will be converted to hexadecimal ASCII and added to *ascii–file*. Pathnames are stripped off;

extensions are left intact. If you use `-l`, the screen names in *ascii–file* will be in lower case.

With `-x`, *ascii–file* is the input file. `bin2hex` extracts each *screen* in *ascii–file* and puts each file in the current directory. If you use `-l`, the screen names are in lower case. Selective extraction of screens from *ascii–file* is not supported. Only one argument is supported with the `-x` option; additional arguments are ignored.

–c
Create an ASCII file from one or more screen.

–x
Extract all screens contained in an ASCII source; selective extraction is not supported.

–f
Overwrite an existing file.

–l
Convert file names sent to output to lower case.

–v
Generate list of files processed.

Converting Screens Between Binary and ASCII

The screen editor creates binary screen files. You can use `f2asc` with `-a` to create an ASCII listing of a screen's contents and edits, modify the file, and convert it back to a binary screen file using `f2asc -b`.

Synopsis

`f2asc -a` *[* `cf` *] ascii–file* `[-i` *header file*`]` *screen...*

`f2asc -b` *[* `f` *] ascii–file*

Options and Arguments

ascii–file
With the `-a` option, *ascii-file* is the name of the file to receive ASCII version of *screen*. With the `-b` option, *ascii-file* is the name of the file to convert into a binary screen.

screen
The file name of a screen to convert to ASCII.

–a
Create ASCII listing of one or more screens.

–b
Create or extract all binary screens from an ASCII listing. Note that this option does not accept an output file name.

–c
Do not generate comment lines (`-a` option only).

–f
Overwrite an existing file.

–i
Include *header file* at beginning of ASCII output.

Description

With `f2asc`, either the `-a` or `-b` option must be used. With `-a`, you must specify the name of at least one screen, (or use wildcard characters). With `-b`, screen names are ignored. The `-b` option automatically extracts all screen files from *ascii–file*.

`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—the widgets that compose it and their respective properties. It is broken into sections by object type, starting with the screen itself, then any groups on the screen, followed by the fields of the screen in numerical order, and finally the labels and boxes on the screen. Each object within the object types begins with its own header:

`S:` *screenname*
`C:` *control-string*
`G:` *groupname*
`T:` *tableviewname*
`F:` *fieldname*
`L:` *labelname*
`B:` *boxname*

Comments appear in lines beginning with the # character. 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=group 1` shows that a field belongs to group 1 of a screen. Value keywords that begin with `PI` describe graphical properties of an object. Other keywords that are specific to JAM add-on products, such as its CASE interface and ReportWriter, can also appear.

Videobiz Database

This appendix describes the database tables in the videobiz database. The following information is listed for each table:

- Column names.
- Data type of each column.
- Length of character columns.
- Status of column detailing whether it is a primary or foreign key and whether it can accept null values.
- Description of the data to be entered into the column.
- Sample entry.

Videobiz Schema

The following tables outline the database tables in the videobiz database. A diagram of the schema appears in Figure 45 on page 577.

Table 59. Actors table.

Column Name	Data Type	Length	Status	Sample	Description
actor_id	integer		primary key not null	87	Unique number code for each actor.
last_name	char	25	not null	Ullmann	Actor’s last name or only name.
first_name	char	20		Liv	Actor’s first name.

Table 60. Codes table.

Column Name	Data Type	Length	Status	Sample	Description
code_type	char	32	primary key not null	genre_code	Type of code. Corresponds to column name.
code	char	4	primary key not null	ADV	Code value.
dscr	char	40		Adventure	Description of code value.

Table 61. Customers table.

Column Name	Data Type	Length	Status	Sample	Description
cust_id	integer		primary key not null	2	Unique number code for each customer.
last_name	char	25	not null	Scott	Customer's last name.
first_name	char	20	not null	Alexander	Customer's first name.
address1	char	40		5601 Wilson	Customer's address.
address2	char	40			Additional address information.
city	char	25		Geneva	City customer lives in.
state_prov	char	10		NY	State/Province.
postal_code	char	10		10234	Postal code.
phone	char	15		515–221–4111	Customer's telephone number.
cc_code	char	4		VISA	Code for type of credit card. List in `codes` table.
cc_number	char	16		4000...	Number on credit card.
cc_exp_month	integer			2	Month of credit card expiration. 1=January, 12=December.
cc_exp_year	integer			1994	Year of credit card expiration (4 digits).
member_date	datetime			1991/05/30 00:00:00	Date when customer became a member.
member_status	char	1	not null	A	Current status of membership. Values include: (A)ctive, (I)nactive, (F)requent renter.
num_rentals	integer		not null	105	Total number of rentals customer has made.
rent_amount	float		not null	175.00	Total amount of money paid by customer.
notes	char	254		Likes ADV videos.	Comments about customer.

Table 62. Flag table.

Column Name	Data Type	Length	Status	Description	Sample
yesno	char	1		Flag used in the sample application.	Y

Table 63. Pricecats table.

Column Name	Data Type	Length	Status	Sample	Description
pricecat	char	1	primary key not null	N	Unique letter code for each category.
pricecat_dscr	char	40		New Release	Category description.
rental_days	integer		not null	2	Number of rentals days available in this category.
price	float		not null	2.50	Amount to be paid for rentals in this category.
late_fee	float		not null	2.00	Amount of late fee for rentals in this category.

Table 64. Rentals table.

Column Name	Data Type	Length	Status	Sample	Description
cust_id	integer		primary key foreign key not null	3	Code identifying the customer for this rental.
title_id	integer		primary key foreign key* not null	69	Code identifying the video title for this rental.
copy_num	integer		primary key foreign key not null	2	Copy of this video being rented.
rental_date	datetime		primary key not null	1993/10/29 19:56:00	Date/time the video was rented.
due_back	datetime		not null	1993/11/01 00:00:00	Date the video is due back to avoid late fee.
return_date	datetime			NULL	Actual date/time the video was returned; NULL until then.
price	float		not null	3.50	Rental fee for video at time rental was made.
late_fee	float		not null	1.00	Late fee per day for video at time rental was made.
amount_paid	float		not null	3.50	Total amount paid on this rental as of current date.
rental_status	char	1	not null	C	Status of rental. Values include (C)urrently out, Back and (P)aid, (B)alance is due.
rental_comment	char	76		NULL	Comments about rental, if any.
modified_date	datetime		not null	1993/10/29 19:56:00	Date this record was last modified.
modified_by	integer		foreign key not null	2	Last user who modified record.

*title_id is a foreign key from the tapes table, in combination with copy_num.

Table 65. Roles table.

Column Name	Data Type	Length	Status	Sample	Description
title_id	integer		primary key foreign key not null	33	Unique number code for each video title.
actor_id	integer		primary key foreign key not null	87	Unique number code for each actor.
role	char	40		Marianne	Role the actor plays in the video.

Table 66. Tapes table.

Column Name	Data Type	Length	Status	Sample	Description
title_id	integer		primary key foreign key not null	33	Unique number code for each video title.
copy_num	integer		primary key not null	1	Number identifying the copy of this video.
status	char	1	not null	O	Code specifying the current status of this copy. Values include (A)vailable, (R)eserved, (O)ut, (I)nactive.
times_rented	integer		not null	53	Number of times this copy has been rented.

Table 67. Titles table.

Column Name	Data Type	Length	Status	Sample	Description
title_id	integer		primary key not null	33	Unique number code for each video title.
name	char	60	not null	Scenes from a Marriage	Video title.
genre_code	char	4		CLAS	Code specifying the video category. Values include: ADLT, ADV, CHLD, CLAS, COM, HORR, MUS, MYST, SCFI, TV, VID. See `codes` table.
dir_last_name	char	25		Bergman	Director's last name.
dir_first_name	char	20		Ingmar	Director's first name.
film_minutes	integer			168	Length of the video.
rating_code	char	4		PG	Rating code given the film by the Motion Picture Association of America. Values include: G, PG, PG13, R, NC17. See `codes` table.
release_date	datetime			1974/01/01 00:00:00	Year the film was released to movie theatres.
pricecat	char	1	foreign key not null	G	Code taken from the `pricecats` table specifying the price category.

Table 68. Title_dscr table.

Column Name	Data Type	Length	Status	Sample	Description
title_id	integer		primary key foreign key not null	33	Unique number code for each video title.
line_no	integer		primary key not null	1	Line number of the video description.
dscr_text	char	76		Relationship of a couple...	Description of the video.

Table 69. Users table.

Column Name	Data Type	Length	Status	Sample	Description
user_id	integer		primary key not null	3	Unique number code for each employee/system user.
logon_name	char	8		jack	User's logon name.
password	char	8		forest	User's password.
last_name	char	25		Ryan	User's last name.
first_name	char	20		Jack	User's first name.
customer_flag	char	1		Y	Y allows access to customer subsystem.
admin_flag	char	1		N	Y allows access to administrative subsystem.
marketing_flag	char	1		Y	Y allows access to marketing subsystem.
frontdesk_flag	char	1		Y	Y allows access to front desk subsystem.

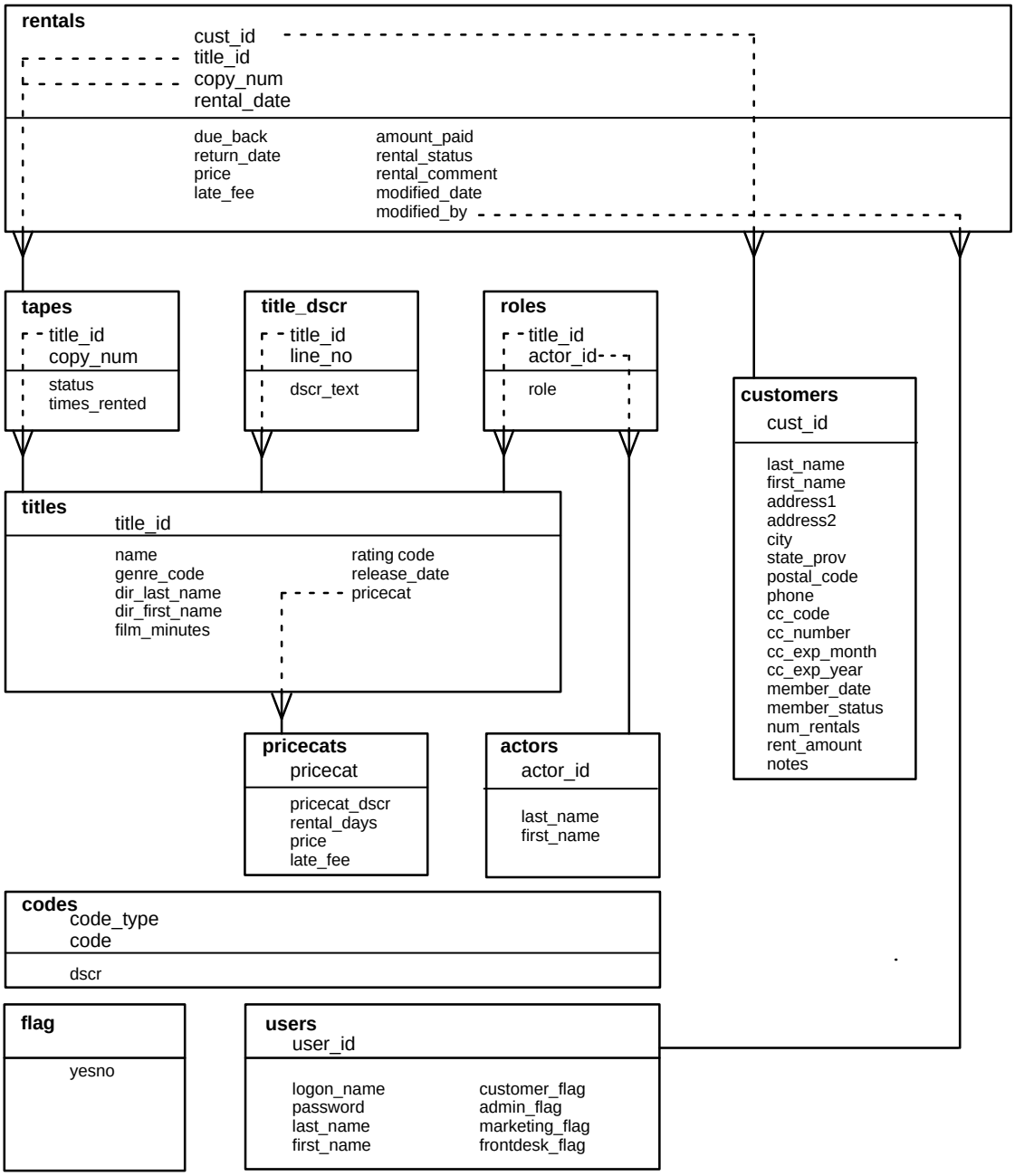
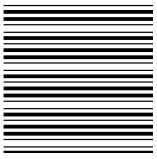


Figure 45. Diagram of the videobiz database.



Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y