JAM

VOLUME I

- Overview
- Author's Guide
- Configuration Guide
- Utilities Guide
- Appendices
- Index

The Composer for Sophisticated Applications.

JAM®
Release 5.03

JYACC Application Manager

November 20, 1992

This is the manual for **JAM**® Release 5.03. It is as accurate as possible at this time; however, both this manual and **JAM** itself are subject to revision.

Please send suggestions and comments regarding this document to:

Technical Publications Manager
JYACC, Inc.
116 John Street
New York, NY 10038

(212) 267–7722

Master List of Contents

The **JAM** Release 5.03 Manual is printed in two volumes. Each volume comprises several parts. Tabbed separators are provided to simplify access to the major parts. The parts are listed below.

Volume I

Overview Tab:

JAM Development Overview
New Features

Author's Guide Tab:

Author's Guide

Configuration Guide Tab:

Configuration Guide

Utilities Guide Tab:

Utilities Guide

Appendices Tab:

Glossary
Upgrade Guide

Index Tab:

Master Index for: **JAM** Development Overview, Author's Guide, Configuration Guide, Utilities Guide, Glossary, Upgrade Guide, JPL Guide, and Programmer's Guide

Volume II

JPL Guide Tab:

JPL Guide

Programmer's Guide Tab:

Programmer's Guide

JAM Development Overview

TABLE OF CONTENTS

Chapter 1
Introduction

This document is intended for new **JAM**® developers, or **JAM** developers who want to get a better understanding of the product. This document is intended to provide a conceptual overview of **JAM**. **JAM** is a powerful tool for prototyping and developing applications, and is structurally unlike other tools available. This document will help you better understand and exploit **JAM**.

JAM is part of a family of JYACC products. The following table describes the rest of the family:

Product	*Description*
JAM/DB*i*	Interface for SQL relational database systems
JAM/DB*i* ReportWriter	Report writer for **JAM**/DB*i*
JAM/P*i* for Motif	Presentation interface for the Motif GUI
JAM/P*i* for Microsoft Windows	Presentation interface for Microsoft Windows
JAM/P*i* for Graphics	Presentation interface for Graphics
Jterm	Color Terminal Emulator optimized for **JAM**

Chapter 2
What is JAM?

JAM is a software toolkit that aids developers in prototyping and building applications with sophisticated user interfaces. Although there are some data manipulation algorithms built into JAM, its primary use is building those application components that interact with end–users and the computer. JAM has been used in applications ranging from order entry and customer information systems to expert systems for project planning to real–time manufacturing applications.

In this chapter, we look briefly at the components of JAM.

2.1
COMPONENTS OF THE JAM ARCHITECTURE

In the following sections, we briefly discuss the architectural components of JAM that work together in a JAM application.

2.1.1

JAM's Core: The Screen Manager

JAM's main concern is the user interface. The fundamental building blocks for user interfaces are screens. Consequently, the core of the JAM product is the *Screen Manager*, a set of C library routines[1] accessible to the programmer for the display and manipulation of screens and screen field data[2].

The Screen Manager is also responsible for interpreting user input to the application, whether it be data input, function key and menu selections, or other keys that move the cursor through the fields on a screen. Sometimes the Screen Manager will process user input; other times it will return to the calling program or call a developer–written function. Generally, input pertaining to a given screen will be processed by the Screen Manager either directly or by calling developer–written functions we designate *hook functions*, while input that causes an interaction between screens will be returned by the Screen Manager and processed at a higher level.

Very sophisticated user interfaces can be created with the Screen Manager alone. However, the fundamentally unique component in JAM's toolset is the JAM Executive described in the next section.

The JAM Screen Manager operates on *screen binaries* which are data structures created and maintained by the JAM Screen Editor (section 2.3.1 on page 10). These structures specify all of the information the Screen Manager needs to do its job, including the size of the screen, the positions and attributes of the fields, the restrictions on user input, etc.

2.1.2

JAM's Controller: The JAM Executive

The JAM *Executive* is a routine supplied with JAM that uses data in screens to control the flow of an application from screen to screen. This eliminates the need to write C code to control the application flow. The supplied executive algorithm is flexible enough to allow the construction of virtually any application. By using the JAM

1. JAM is written in C, and is distributed with the C programming interface. Developers who prefer to use Fortran, Cobol, or PL/1 can, on some platforms, obtain library interface modules for their language of choice from JYACC. In this documentation set, we will often speak of the C language when referring generally to supported third–generation languages as a group.

2. Screens, fields, and other screen components will be discussed in detail at a number of other points in this documentation. It is sufficient at this point to understand that a screen is a rectangular region displayed on the user's display terminal, and that screens often have fields on them for the entry and display of application data.

Executive, you can begin to prototype and build applications without doing any low–level programming.

Like the Screen Manager, the JAM Executive can be directed to call developer–written C functions during an application. Unlike the Screen Manager, though, it is not a collection of routines. The JAM Executive is a single routine which invokes the Screen Manager in ways that allow application control information to reside in screens.

Just as the Screen Manager hides screen details from the calling programs, the JAM Executive hides details of screen interactions from the applications. Quite often, these details may change, resulting in a substantially different user interface without affecting any application code. This means that when user interface specifications change in a project, the application code may not need to be altered.

It is possible to write applications that do not use the JAM Executive. In this case, you will need to write your own executive from scratch. Only in rare cases will this be necessary; the JAM Executive is more powerful than it may appear at first glance. Most of the documentation will assume the use of the JAM Executive. In this document, the words "JAM application" often mean "an application using the JAM Executive".

For a brief discussion of deciding when to write your own executive, please see section 5.2 on page 45.

2.1.3
Screen Manager/JAM Executive Interaction

It is critical for JAM Programmers to understand the relationship between the JAM Executive and the Screen Manager. The JAM Executive provides the glue that links the screens of the application together by directing the Screen Manager to display screens in a particular order. The Screen Manager is responsible for actually displaying screens and opening a dialogue with the user on those screens (i.e. opening the keyboard for user input). The Screen Manager moves the cursor, modifies the display, and processes data entered and keys struck by the user. When a function key is struck or a menu selection is made, the Screen Manager immediately returns control to the JAM Executive along with information about which function key or menu selection was chosen. The JAM Executive then peeks at the screen itself to determine what action is associated with the function key or menu event. If the indicated action is that a new screen should be opened, the JAM Executive directs the Screen Manager to do so and the cycle starts anew. If some other action is indicated, the Executive takes that action and then directs the Screen Manager to re–open the dialogue with the user on the same screen. This process is shown in Figure 1.

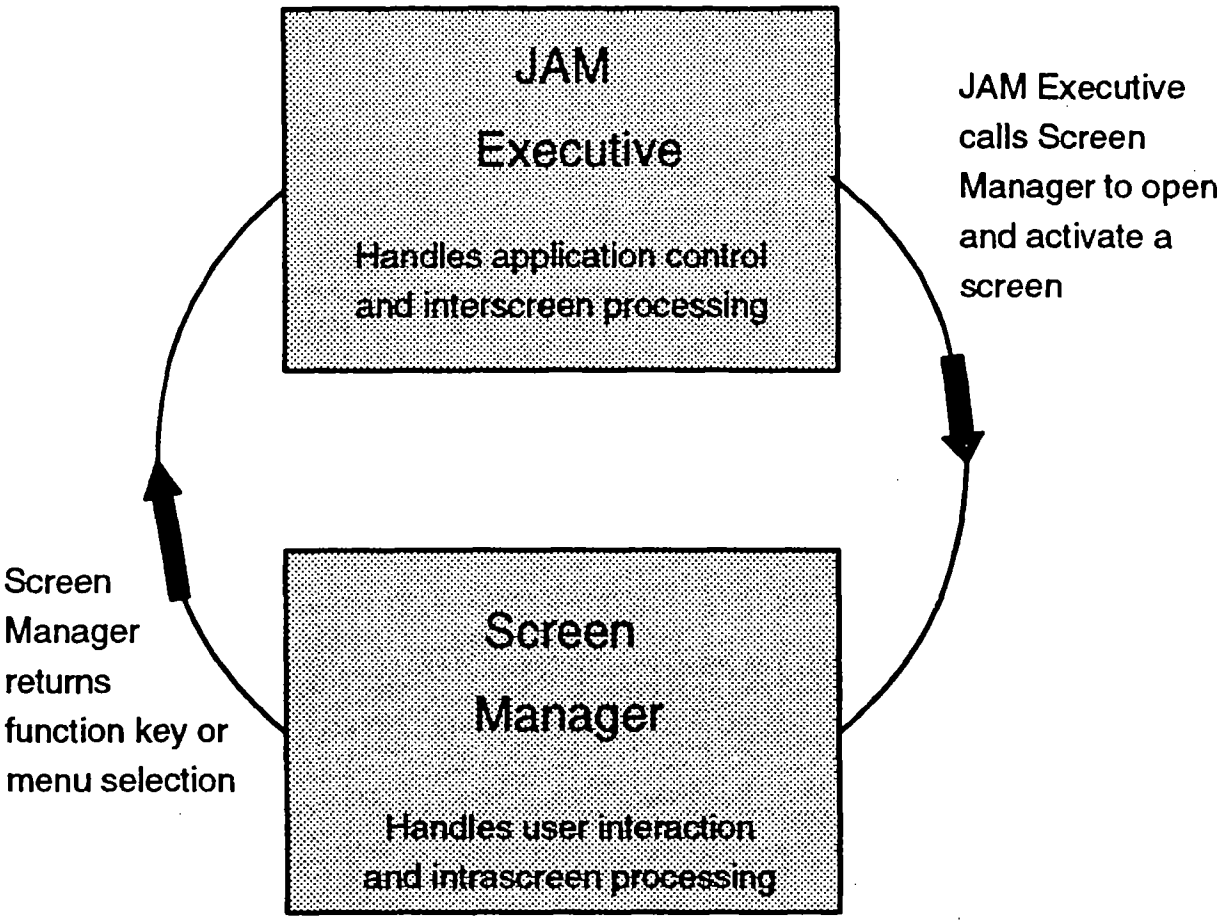


Figure 1: Simple Schematic Model of the **JAM** Screen Manager/Executive Interaction.

2.1.4

The Local Data Block

The *Local Data Block*, or LDB, is another important feature of **JAM**. It is a set of routines and structures used to share application field data among screens.

The LDB is created at runtime from the *data dictionary*, a file of LDB fields and their characteristics. The data dictionary is created with the Data Dictionary Editor (section 2.3.2 on page 11). Aside from driving the creation of the runtime LDB, the data dictionary can be used in a number of ways during development to ensure the consistency of

fields across screens in an application, and to link the data in an application with existing data bases or other data sources.

Every time a screen is made active by the Screen Manager at runtime, its fields are populated with data from associated fields in the LDB. When the screen is closed or made inactive, all its fields are copied to their associated fields in the LDB. This eliminates the need to write code to move data from screen to screen, and allows application hook functions to access field data regardless of whether or not it is on the active screen.

2.1.5
JPL

JAM comes with an interpreter for the *JYACC Procedural Language*, or JPL. JPL is a procedural language, often preferable to C for simple field validations and data manipulations[3]. The JPL syntax supports calls to C functions — these can be developer–written hook functions or functions from **JAM**'s libraries. JPL is fully described in the *JPL Guide.*

2.2
COMPONENTS OF JAM

The **JAM** product is packaged as a number of programs and data files. In the sections that follow, we briefly discuss the main components of the product and how they are used. For a file by file description, please see the Installation Guide.

2.2.1
The Authoring Tool

The authoring tool `jxform` is the core development tool provided with **JAM**. It allows developers to run **JAM** applications just as an end user might, and also provides direct access to the Screen Editor, the Keyset Editor, and the Data Dictionary Editor. It is described in detail in the *Author's Guide.*

2.2.2
JAM Libraries

Several runtime libraries are supplied with the product. These libraries include the following:

3. Developers who are also using **JAM/DBi** can embed SQL queries for accessing their relational database directly into their JPL code.

- The Screen Manager routines. The Screen Manager libraries include the JPL support routines.
- The JAM Executive routine and its supporting routines. The JAM Executive libraries include the LDB routines.
- The authoring support routines[4]. These allow developers to create a new versions of the authoring tool linked with developer–written hook functions.

2.2.3

Source Code

JAM supplies some source code with the product that may be used and/or modified. There are two main source modules: `jxmain.c` is used for compiling and linking new versions of the authoring tool and `jmain.c` is used for compiling and linking JAM Application Executables.

2.2.4

Screens

To accommodate natural language localization requirements, all the screens and help screens used by the authoring tool are supplied so that they can be customized.

2.2.5

Configuration Files

Several default configuration files are included. These files are used by JAM to operate properly with the video display hardware, your keyboard, and other aspects of a hardware and software environment. For more information, please see the *Configuration Guide*.

2.2.6

JAM Utilities

These utilities are used for a variety of tasks that make the development and maintenance of JAM applications and JAM configuration files easier. The utilities are documented in detail in the *Utilities Guide*.

4. Please note that you are only licensed to use the authoring support runtime libraries to create new versions of the authoring tool that incorporate functions you have written yourself. You are not licensed to distribute applications linked with this library.

There are three utilities that may prove particularly important to developers. `f2asc` transfers screens back and forth between binary and ASCII representations. `dd2asc` does the same thing for data dictionaries. The `jamcheck` tool is used to ensure that all the fields in a particular application are consistent with each other and with the data dictionary entries. In addition to flagging potential inconsistencies, `jamcheck` can globally propagate changes made to a single field across all the screens in an application.

2.3
COMPONENTS OF A JAM APPLICATION

JAM applications generally have the five components discussed in the following sections: screens, data dictionaries, JPL modules, developer–written C hook functions, and the **JAM** Application Executable. These are illustrated in Figure 2.

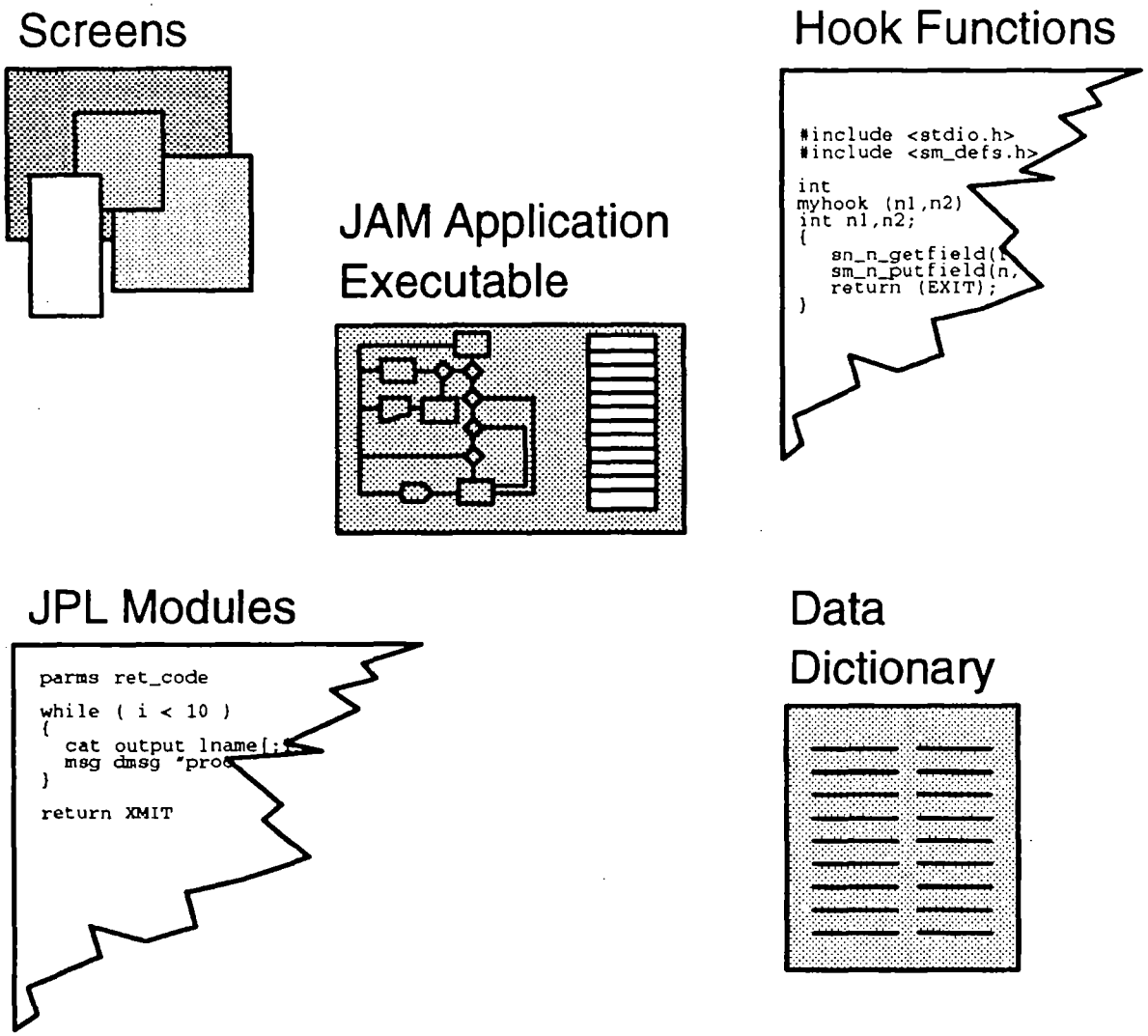


Figure 2: Components of a **JAM** Application.

2.3.1

JAM Screens

Screens are the fundamental building blocks of **JAM** applications. They contain the information the Screen Manager needs to display them as well as the control information used by the **JAM** Executive to control the flow of the application. Screens are created and modified with the Screen Editor which is part of the main authoring tool `jxform`, and are stored as *screen binaries*. Screen Binaries are the static data represen-

tations of screens used by the Screen Manager at runtime to build and display a screen on the display terminal. Screen binaries are usually files, but may be turned into C language data structures and compiled into the JAM Application Executable, or combined into screen libraries.

Screens are rectangular display areas, and are not limited in size by the display terminal. Depending on the Screen Manager routine used to display a screen, it can be displayed as a *form* or as a *window*. When a screen is displayed as a form, all previously open screens on the display are closed. When a screen is displayed as a window, it is overlaid on top of whatever screens may already exist on the display. Therefore, there can be at most one form displayed at a given time, but any number of windows may be stacked on one another. A screen can contain hooks to call C or JPL routines when the screen is opened or closed.

Screens contain constant display text and fields for application data entry and display. Fields may be designated for data entry, as individual selections in a menu, or as members of radio button groups and check–list groups.

Fields may be assigned a variety of characteristics. These include identifying names, edit masks restricting the field to certain kinds of data, display attributes, and hooks to C or JPL routines to call when the field is entered, exited or validated.

Application control flow information is stored in the screens as *control strings*. Control Strings are associated with either menu selections or function keys. Control strings can display (and transfer control to) other screens, execute operating system commands or programs, or invoke C or JPL routines.

2.3.2
The Data Dictionary

The *data dictionary* is used to create the LDB when a **JAM** application starts. It is a list of entries that are associated by name with screen fields. All of the characteristics that can be associated with screen fields can be designated also for data dictionary entries. The data dictionary is commonly stored in a binary file called `data.dic`, and is created and maintained with the Data Dictionary Editor linked with the `jxform` utility. The Screen Editor also has the capability to place screen fields as entries into the data dictionary and to create screen fields from entries in the data dictionary.

2.3.3
C Hook Functions

Both the Screen Manager and the **JAM** Executive can invoke developer–written C functions. These functions must be compiled and linked into the **JAM** Application Executable and properly installed. See the *Programmer's Guide* for details.

2.3.4

JPL Modules

A JPL module is a collection of JPL procedures. JPL is interpreted and can be stored as ASCII text or as compiled[5] binary data in JPL modules. A JPL module can be any of the following:

- Disk–based, stored in a file,
- Memory resident, linked in to the JAM Application Executable, or
- Screen resident, attached to a screen or particular screen field.

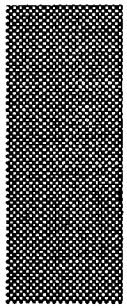
JPL code can be written with a text editor or entered into the screen binary with the Screen Editor.

2.3.5

JAM Application Executable

The executable program that drives your application is called the **JAM** *Application Executable*. It is created by the linker on your machine from the supplied (and possibly modified) main source module `jmain.c`, the Screen Manager library, the **JAM** Executive, and all the C hook functions written to be invoked as callbacks by the **JAM** Executive or the Screen Manager. The **JAM** Application Executable is your application program. It generally contains little control flow information. The control information is found on the screens.

5. Compiled JPL is not native machine code, but rather a tokenized and syntax–checked binary version of the JPL source.



Chapter 3
JAM *Application Development*

Developers prototyping and building **JAM** applications use the authoring tool, `jxform`. This utility is a **JAM** Application Executable linked with the **JAM** Executive, the Screen Manager, the Screen Editor, and the Data Dictionary Editor. At the top level (application mode), `jxform` navigates through the application just as the runtime executable will for the end–user. However, `jxform` also provides direct access to the Screen Editor and the Data Dictionary Editor which allows the developer to create or modify key parts of the application while simulating and testing the end–user experience. When a particular screen is active, the developer can immediately enter the Screen Editor, modify that screen, test it, and return to it in application mode at the same spot in the application. Control strings are created and modified in the Screen Editor in order to affect and control application flow. Similarly, the developer can edit the data dictionary and continue running the application using the new file. JPL modules can also be edited on the fly.

The only things that cannot be changed from within the `jxform` utility are the developer–written C functions linked to the authoring tool. To use new developer–written C functions, the tool must be re–linked and re–started. Linking developer–written code into the authoring tool or into the **JAM** Application Executable is discussed in detail in the *Programmer's Guide*.

3.1
CREATING AND EDITING APPLICATION SCREENS

Generally, a developer starts creating a new application by creating screens with the Screen Editor. Screens contain display information, data entry regions, and application

control information. The creation and subsequent development of screens is the bulk of the authoring task.

Once in the Screen Editor, the developer can specify some characteristics for the screen itself. Screens have dimensions, colors and borders. In addition, the developer can specify the names of C or JPL hook functions to be called when the screen is opened or closed.

Display data is added to the screen as text or graphics characters. Fields are placed on the screen by typing underscores. Characteristics can then be associated with the fields. There are approximately fifty modifiable field characteristics including field names, field edits, next and previous fields, display attributes, and help text. In many cases, the default characteristics are sufficient. C or JPL hook functions can be attached to specific fields and invoked on field entry, exit, or validation. Fields can be made into horizontal or vertical *arrays*, and any array can be designated to be able to hold more data occurrences than are actually visible onscreen at any given point in time. Since the end–user can scroll forward and backward through such arrays, they are called *scrolling arrays*.

Fields and display data can be moved and/or copied around the screen using the Screen Editor, either individually or in selected blocks. Fields can also be copied into the data dictionary while in the Screen Editor, and the data dictionary can similarly be used to populate the screen with fields. This is useful for the propagation of consistent fields on screens throughout the development of the application, and for faithful propagation of field data from screen to screen at runtime.

The developer will compose *control strings* and associate them on the screen he is editing with menu selections or function keys. The **JAM** Executive interprets control strings at runtime to:

- Open and transfer control to additional screens as base forms or windows,
- Execute operating system commands or programs, and
- Invoke JPL or C functions.

The behavior of a screen can be tested from within the Screen Editor to ensure that the fields work correctly. Control Strings, however, must be tested in the top level application mode of the `jxform` utility.

3.2
CREATING AND EDITING THE DATA DICTIONARY

There are four mechanisms for creating or modifying the data dictionary:

- The Screen Editor can be used to add a screen field to the data dictionary, to bring a data dictionary entry into a screen, or to compare the characteristics of a screen field and an associated data dictionary entry.
- The Data Dictionary Editor, invoked from application mode, can be used to add delete, and change data dictionary entries.
- The `dd2asc` utility converts **JAM** data dictionary files between binary and ASCII formats. This facilitates the manipulation of the data dictionary outside of the authoring tool, perhaps to integrate the **JAM** data dictionary with that of a CASE tool or a DBMS.
- The `jamcheck` utility compares screen fields with data dictionary entries. Optionally, it can propagate data dictionary field characteristics through all of the screen fields.

3.3
ITERATIVELY TESTING AN APPLICATION

The authoring tool can switch among the Screen Editor, the Data Dictionary Editor, and application mode, where the developer simulates end–user runtime experience. Therefore, problems with screens, the data dictionary, and JPL code can be fixed in–line and immediately re–tested without need for a time–consuming compile– link–test cycle. Only developer–written C functions will need such a cycle[6]. Such functions are always linked into the runtime application executable and may be linked into the authoring tool for complete application testing and development.

6. If you are using **JAM/DBi**, SQL queries may be tested from application mode.

Chapter 4
JAM *Control Flow*

In this chapter, we discuss the algorithms employed by the Screen Manager and by the JAM Executive, and how the two interact with each other in a JAM Application Executable. We go into substantial detail in describing the model; it may not be necessary for the reader to fully understand all aspects of this chapter before embarking on a JAM project.

4.1
KEYBOARD AND VIDEO TRANSLATION

All user input to a JAM application is processed through a keyboard translation table before being handled by the Screen Manager. Similarly, all JAM output to the physical display monitor is processed through a video mapping table first. This is shown schematically in Figure 3.

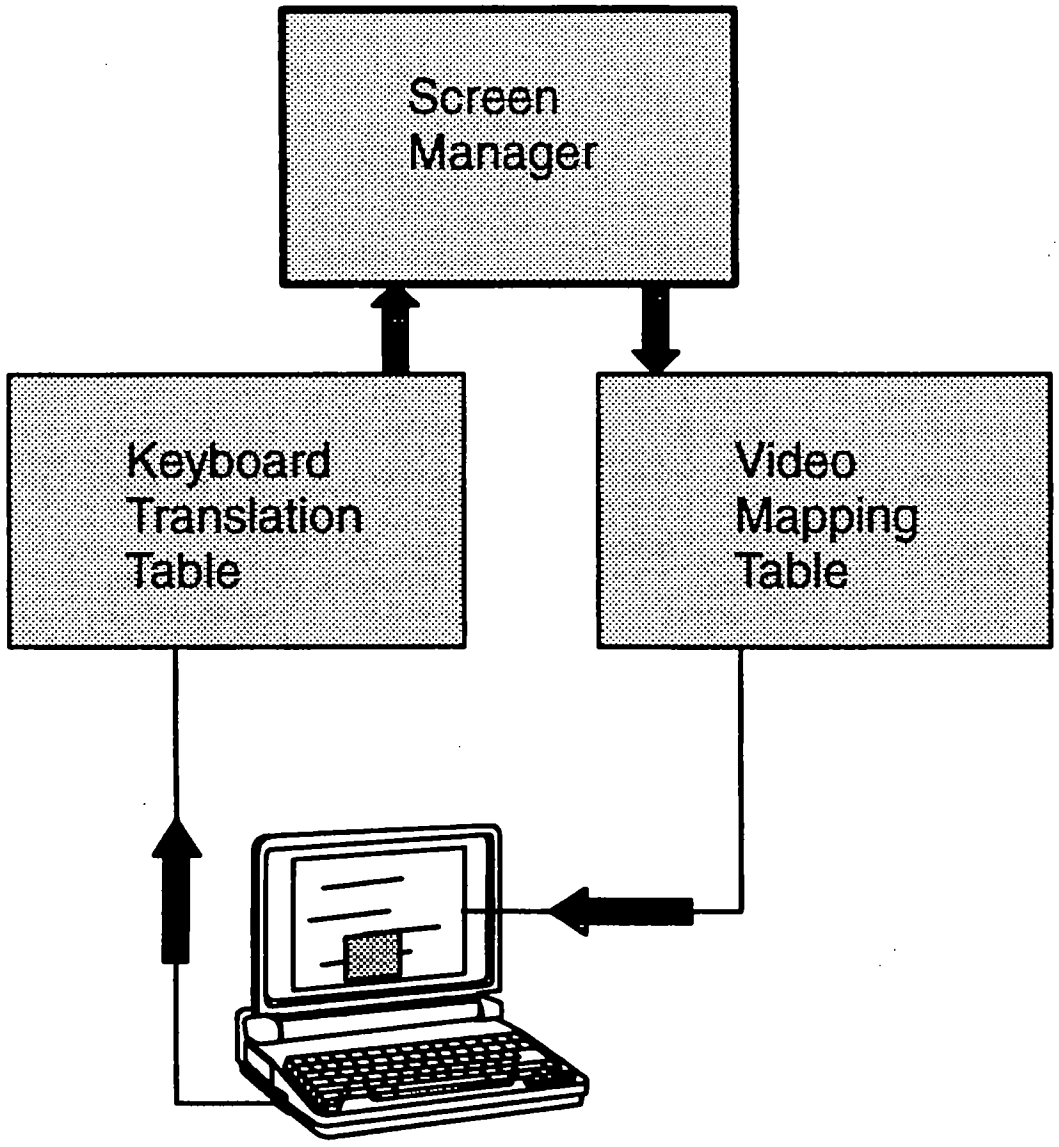


Figure 3: Keyboard and Video Translation.

This translation is done to preserve terminal independence. **JAM** applications can run on a variety of terminals, provided that the appropriate keyboard and video configuration files are created. These configuration files are used by the application at initialization to create the keyboard and video translation tables. The creation of these files is discussed in the *Configuration Guide*.

Physical keystrokes on the terminal are translated into **JAM** logical keys. Thus, the Screen Manager only interprets keys that are members of the **JAM** logical keyboard, discussed in detail in the *Author's Guide*. Of particular interest to our discussion are the **JAM** logical function keys, which are named PF1 through PF24, SPF1 through SPF24, and APP1 through APP24. These function keys, shifted function keys, and application functions keys cause the Screen Manager to terminate the user dialogue and to return to the **JAM** Executive. The EXIT and TRANSMIT keys also terminate the dialogue, but in addition they have some default actions associated with them.

4.2
SAMPLE PERSONNEL APPLICATION — USER'S VIEW

Here we describe an example personnel application to facilitate our discussion of **JAM**. Our example consists of the personnel part of a human resources application. The user can search a simple data base of employees, update the record for a given employee, and look at a salary history for that employee.

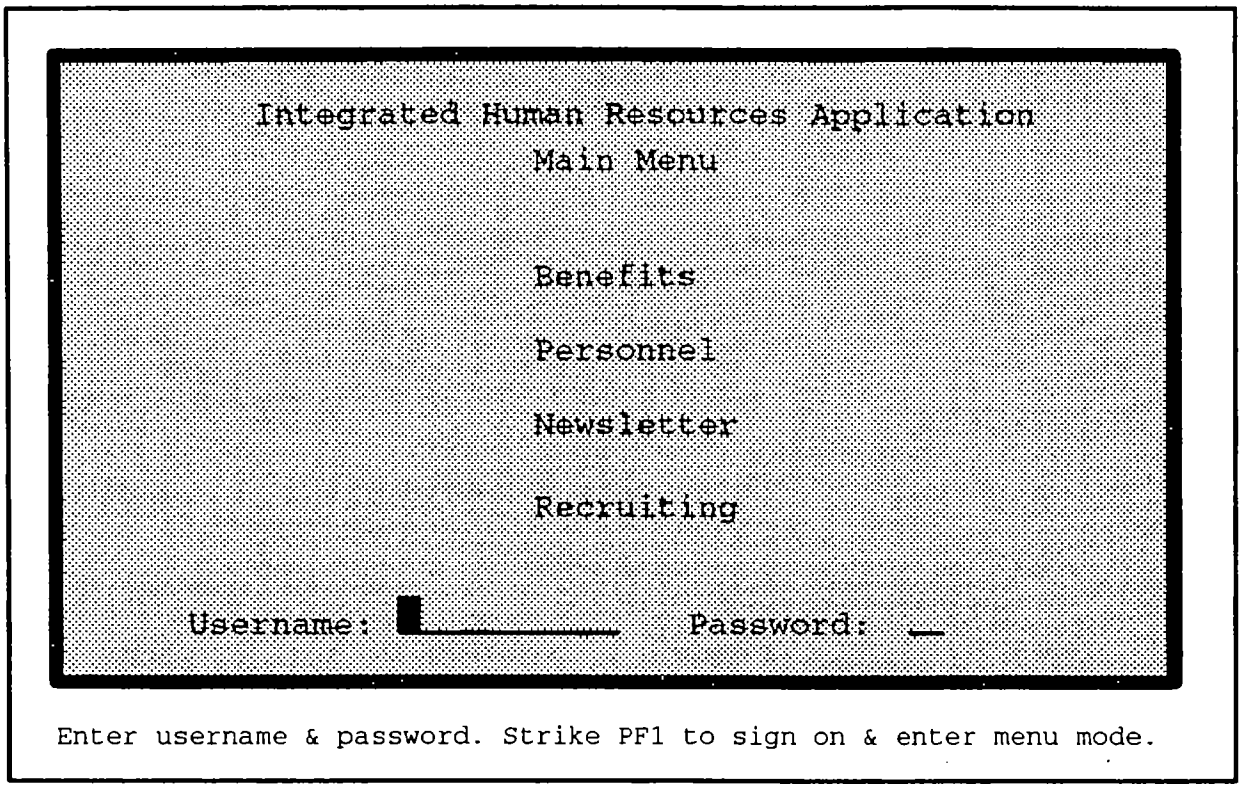


Figure 4: Human Resources Application Main Menu, `mainscrn`.

The first screen of the application is shown in Figure 4. This screen, named `mainscrn`, has data entry fields for a username and a password, and a menu of sub–applications relevant to human resources. This screen is initially treated as a data entry screen, and the cursor is positioned in the username field.

The user types a username and a password, and strikes PF1 to validate the pair and switch into menu mode. The status line is changed as well, so the screen appears as shown in Figure 5.

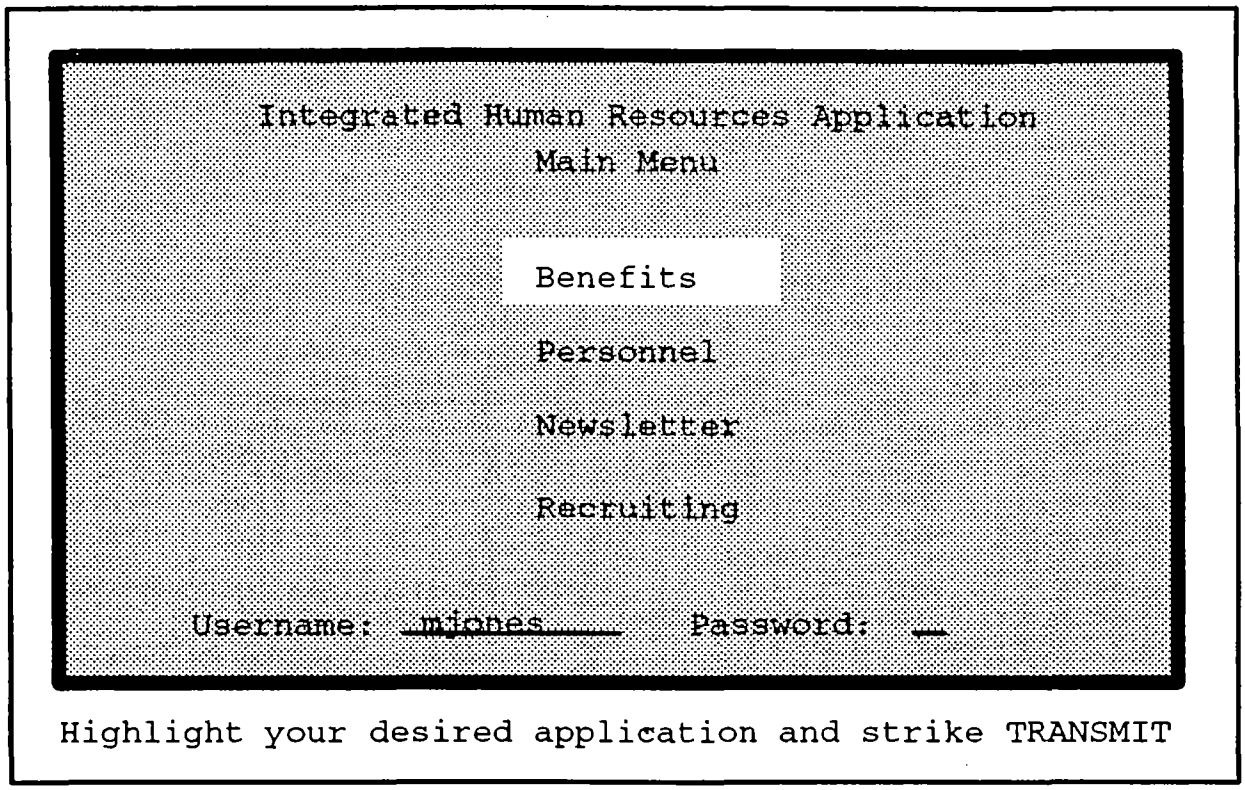


Figure 5: `mainscrn` in Menu Mode.

At any time on this screen, the user can leave the application by striking the EXIT key. The user can enter the personnel application by selecting `Personnel` from the menu. In that case, the employee screen `empscrn`, shown in Figure 6, is displayed as a form.

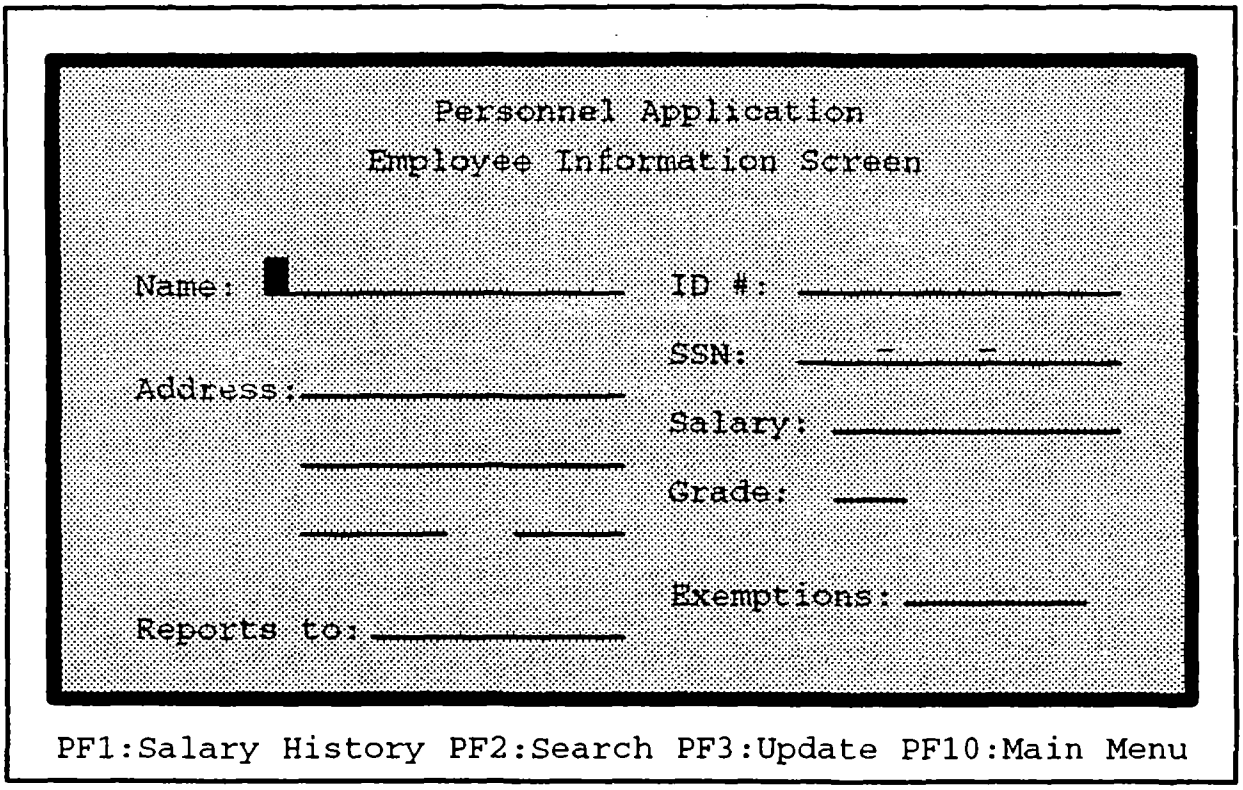


Figure 6: Personnel Application Employee Screen `empscrn`.

On the `empscrn` screen, the user can enter the name, ID number, or any other search criteria for an employee, and strike the PF2 key to search some data base. Successive striking of the PF2 key will browse through employee records that match the entered criteria, unless the user types new information in any field, in which case the search is started again. Striking the PF3 key will update the current employee record with any data the user changed about the selected employee, and the PF10 key returns the user to the Main Menu. Note that the default effect of the EXIT key is also to return to `mainscrn`.

While the `empscrn` screen is active, the user strikes PF1 to display the Salary History screen `salhist` as a pop–up window shown in Figure 7. This screen has detailed information about past salary reviews. The end–user cannot add or change information here, but strikes PF10 to return to the main menu. The EXIT key closes the `salhist` pop–up window and returns to the Employee Screen.

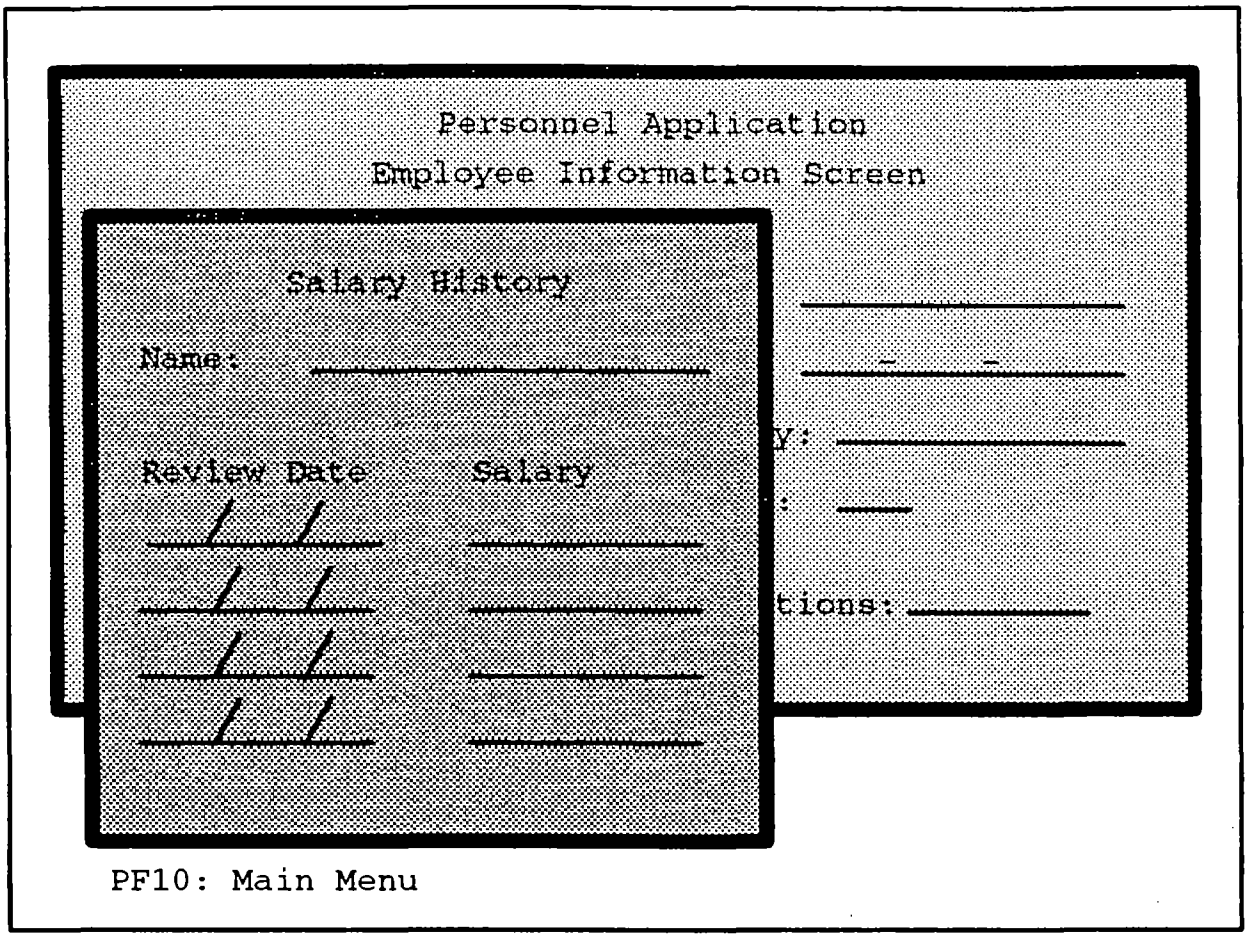


Figure 7: Personnel Application Salary History Window `salhist`.

4.3
THE PERSONNEL APPLICATION — DEVELOPER'S VIEW

As discussed in section 2.3 on page 2.3, a **JAM** application consists of C routines, JPL routines, screens, a data dictionary, and a **JAM** Application Executable. In this section, we briefly discuss how these parts of the Personnel Application are created.

4.3.1
Personnel Application Screens

The application screens are created with the Screen Editor, one of the tools built into the authoring program, `jxform`. The screens `mainscrn`, `empscrn`, and `salhist` are

all created as files with the Screen Editor[7]. A developer's view of `mainscrn` is shown in Figure 8.

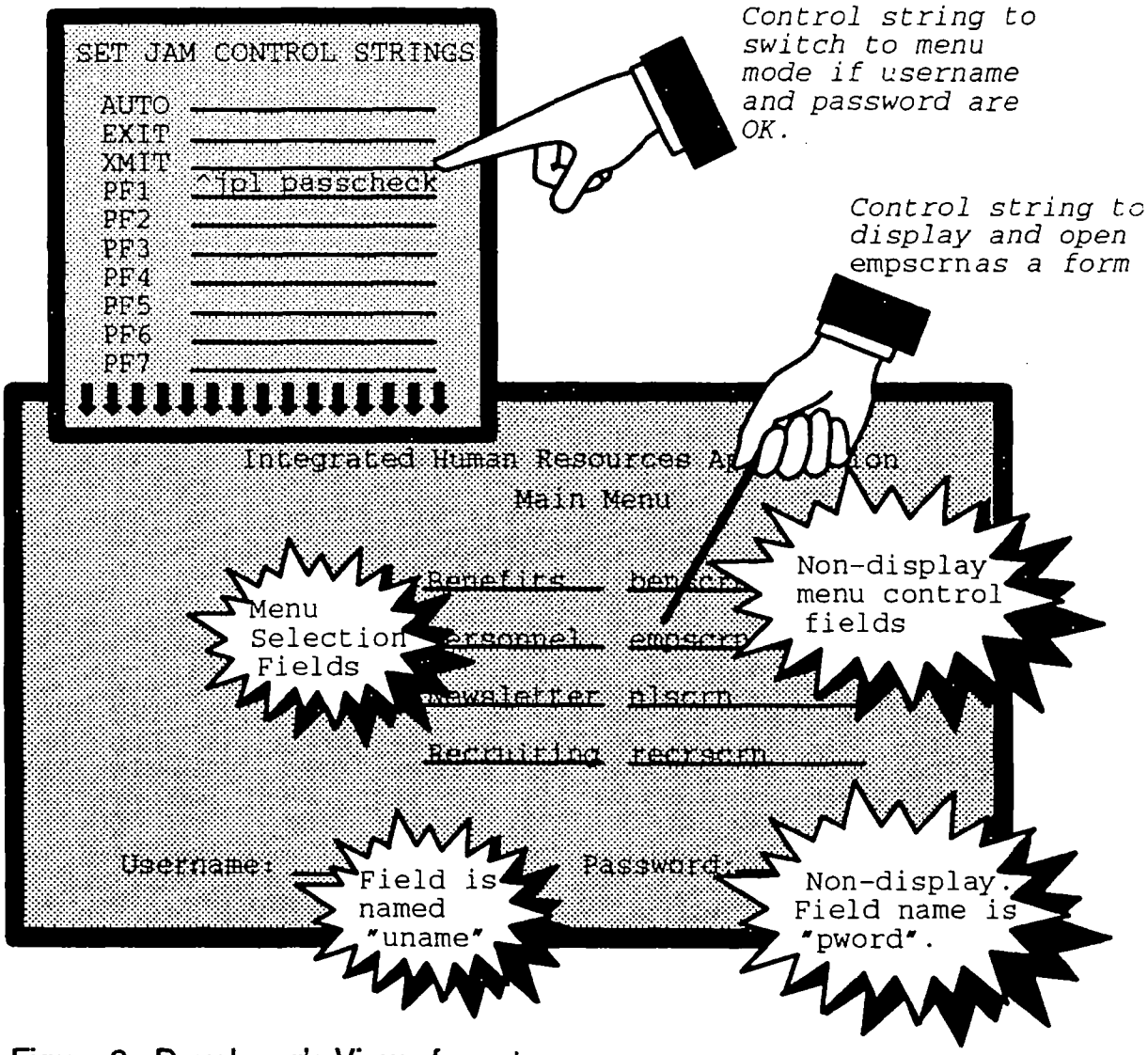


Figure 8: Developer's View of `mainscrn`

The `mainscrn` screen has two data entry fields, one for username and one for password. The username field is named "uname" and has no other distinguishing characteristics. The `password` field is named "pword" and is marked as non–display. There is a menu of four choices in the center of the screen for access to the various applications: `benefits`, `personnel`, `newsletter`, and `recruiting`. To the right of

7. Details of screen creation are found in the *Author's Guide*.

these are four non–display control fields, one associated with each menu selection and protected from entry so that the user is never aware of their presence. The name of the top level screen for each of the respective sub–applications is entered into these control fields. In the case of the `personnel` menu selection, the control string is "`empscrn`". Finally, the screen has a control string associated with the PF1 key to validate the user-name and password fields and to toggle the screen from data entry to menu mode. This control string "`^jpl passcheck`" invokes a JPL routine.

The developer's view of `empscrn` is shown in Figure 9.

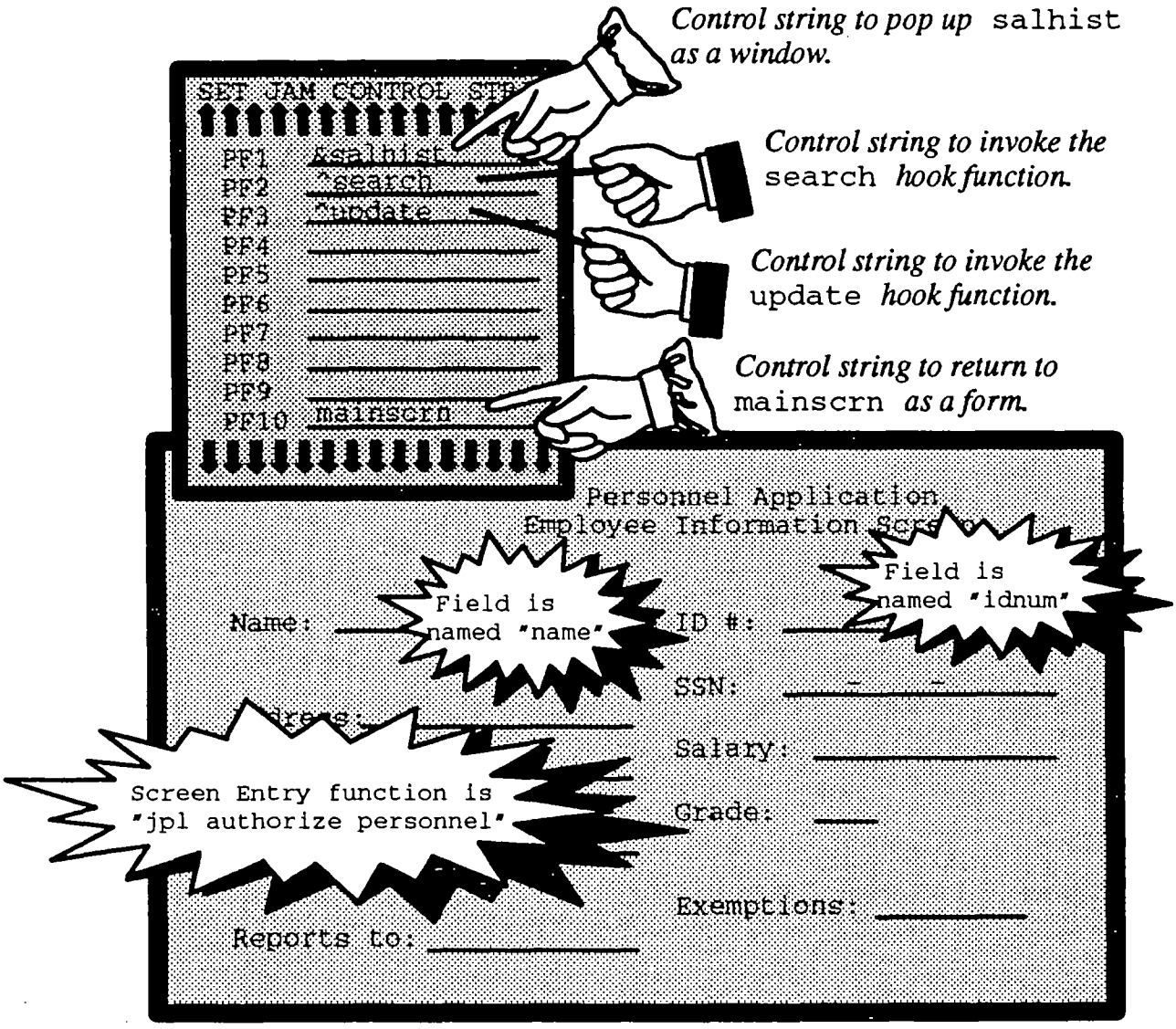


Figure 9: Developer's View of `empscrn`.

Since the `empscrn` screen is the top–level screen for a application, a JPL routine is attached as a screen entry function with the string "`jpl authorize personnel`" to ensure that the user is authorized to use the personnel application. Four function keys are associated with some actions: the PF1 key is given the control string "`&salhist`", the PF2 key is given the control string "`^search`", the PF3 key is given the control string "`^update`", and the PF10 key is given the control string "`mainscrn`".

The developer's view of `salhist` is shown in Figure 10.

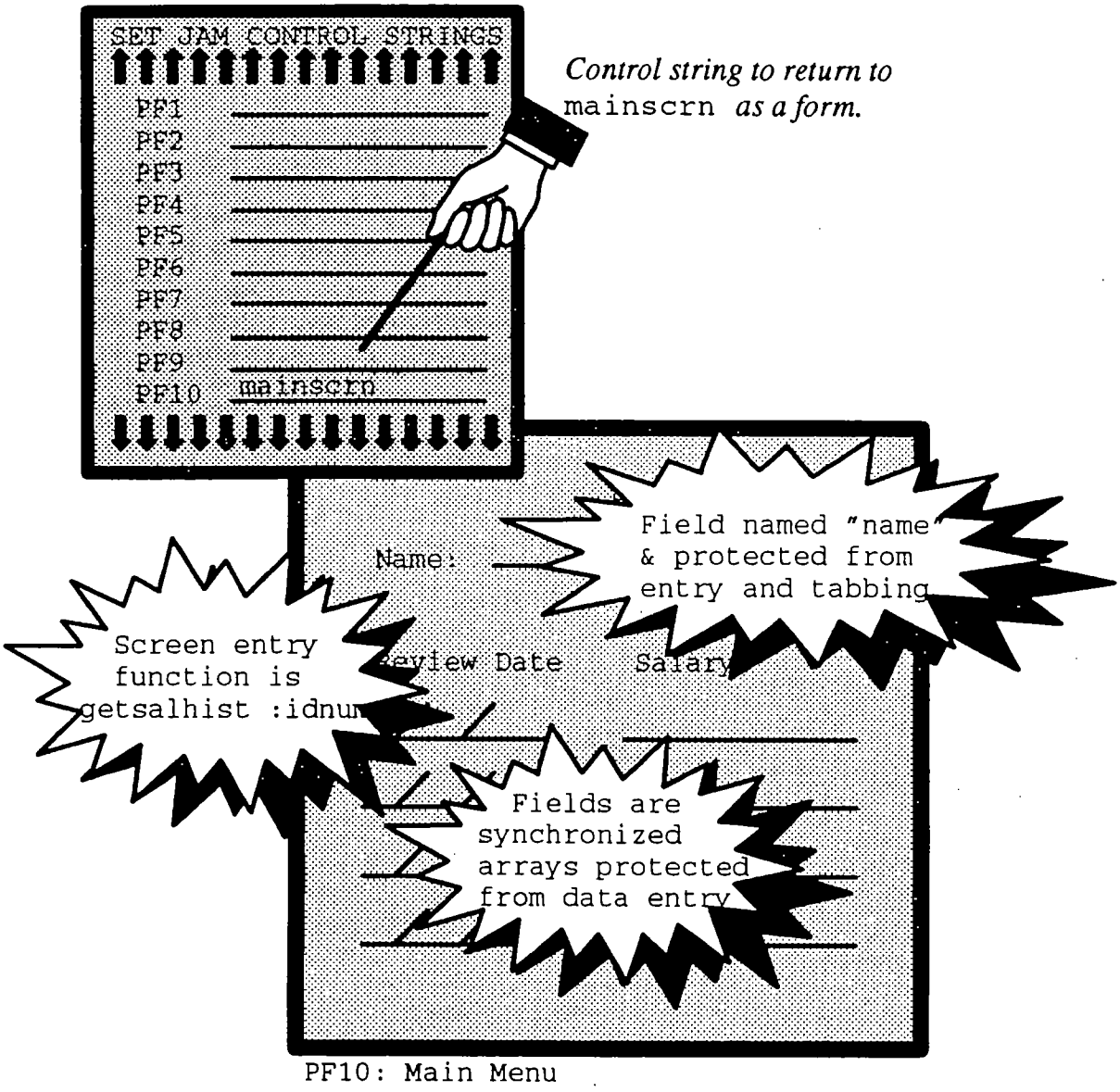


Figure 10: Developer's View of `salhist`.

The `salhist` screen has the employee name field protected so that the user cannot place the cursor there. Review date and Salary are implemented as synchronized scrolling arrays[8] protected only from data entry. One function key is associated with action:

8. Synchronized scrolling arrays are discussed in the *Author's Guide*.

the PF10 key is given the control string "mainscrn". The salhist screen also has a screen entry function specified as getsalhist :idnum[9].

4.3.2

The Data Dictionary

Every data entry field whose value is propagated from screen to screen is in the data dictionary. The fields are placed in the data dictionary either through the Screen Editor, or with the Data Dictionary Editor. Both are described in detail in the *Author's Guide*. Note that every entry in the data dictionary is given a scope. All fields of a given scope can be cleared or initialized at the same time if desired.

A developer's view of the data dictionary is shown in Figure 11.

Data Dictionary

NAME	SCOPE	COMMENT
idnum	2	employee number
name	2	employee name
pword	3	user's password
uname	3	user's name

Figure 11: Developer's View of the Data Dictionary

4.3.3

JPL Modules

Two JPL modules are used in this application. The first one is stored in the screen binary for the mainscrn screen and contains the passcheck procedure. This procedure verifies that the username and password identify a valid user. In addition, passcheck changes the contents of the status line and toggles mainscrn into menu mode.

9. By using colon expansion and function prototyping, both of which are described in the *Author's Guide* and in the *Programmer's Guide*, hook functions can be passed variable parameters in this way.

The second JPL module is stored in a file named `authorize`. It contains the `authorize` procedure, and is used by the top screen of each sub–application to check user authorization for that sub–application.

JPL syntax and scoping rules are described in detail in the *JPL Guide*.

4.3.4
C Functions

Three C functions were written for this application. The first, `search`, searches the underlying database for an employee that matches the criteria placed on `empscrn`. The second, `update`, updates the employee record with new or modified information. The third, `getsalhist`, searches the underlying database for the salary history of the employee with the id number passed as an argument. These C functions must be compiled and linked in with the Screen Manager and the JAM Executive to create a new JAM Application Executable. This executable could have any name. In our example it is named `humres`.

The guidelines for writing, installing, and linking C functions are found in the *Programmer's Guide*.

4.4
THE SCREEN MANAGER / JAM EXECUTIVE DIALOGUE

The application is started by executing the JAM Application Executable from the operating system and passing it the name of the top level screen as an argument. In our example, we would use the following command:

```
humres mainscrn
```

The main program, `jmain.c`, provided in source form with JAM, is linked into the JAM Application Executable, possibly after some modification by the developer (See the *Programmer's Guide*). It initializes the JAM environment using several environment variables and configuration files. The Local Data Block is created from the information in the data dictionary, and LDB entry values are assigned from LDB initialization files if any exist. LDB initialization files are described in detail in the *Author's Guide*. They are used to initially populate the LDB with data.

After initialization is complete, the main program invokes the JAM Executive with the library function call `sm_jtop`, passing, as an argument, the name of the top level

screen in the application, `mainscrn` in our example[10]. The Executive immediately directs the Screen Manager to open the top–level screen as a form with the library function call `sm_r_form`, and then instructs the Screen Manager to activate (start a dialogue with the user on) that screen with the library function call `sm_input`. From that point, the Screen Manager, within `sm_input`, handles user interaction with the screen: moving the cursor, entering data, and providing help. When a function key is struck or a menu choice is selected, the Screen Manager routine `sm_input` returns control to the JAM Executive along with information about what function key or menu choice was selected[11].

The JAM Executive then looks into the screen to find the control string associated with the function key or menu selection. If an invalid control string is found, or if no control string is found, then the dialogue with the user will be re–started with a call to the Screen Manager function `sm_input`. As is described in detail in the *Author's Guide*, control strings can direct the JAM Executive to take one of five actions:

- If the control string begins with a caret character (^), the remainder of the string is interpreted as the name and arguments for an installed developer–written hook function. If the function name is `jpl`, so that the control string starts with the string `^jpl`, the remainder of the string is interpreted as the name and arguments for a JPL procedure. The function can return a function key just as `sm_input` does, in which case the JAM Executive will process the associated control string. Alternatively, the routine can return zero, in which case the JAM Executive will ask the Screen Manager to re–start the dialogue on the currently displayed screen with a call to `sm_input`.
- If the control string begins with an exclamation point (`!`), the remainder of the string is interpreted as an operating system command. A sub–process is spawned and the command is run in that sub–process. Before calling the command, the JAM Executive calls the Screen Manager function `sm_leave` to save the state of the screen and sets the terminal characteristics to the mode expected by the operating system. On return from the command, the JAM Executive calls `sm_return` to reset the terminal characteristics and re–paint the screen. It then directs the Screen Manager to re–start the dialogue on the currently displayed screen with a call to `sm_input`.
- If the control string begins with a single ampersand (&), the remainder of the string is interpreted as the name of a screen to be popped up as

10. If all users of an application have the same top–level screen, the developer may choose to hardcode that name in the jmain.c source code.

11. Note that the JAM Executive makes the Screen Manager calls automatically. For the basic navigation from screen to screen, no C programming is required on the part of the developer.

a window. The Screen Manager is requested to display the window with a call to `sm_r_window`, and then directed to activate that window with a call to `sm_input`.

- If the control string begins with a double ampersand (`&&`), the remainder of the string is interpreted as the name of a screen to be popped up as a sibling window. Sibling windows allow the end–user to cycle through displayed windows without closing them. The Screen Manager is requested to display the window with a call to `sm_r_window`, to make it into a sibling window with a call to `sm_sibling`, and then to activate it with a call to `sm_input`.
- Any control string that does not begin with any of these special characters is interpreted as the name of a screen to be displayed as a form. The Screen Manager is requested to close all open windows and forms and open the named form with a call to `sm_r_form`. It is then directed to activate that form with a call to `sm_input`.

The algorithm for **JAM** Control Flow is shown schematically in Figure 12.

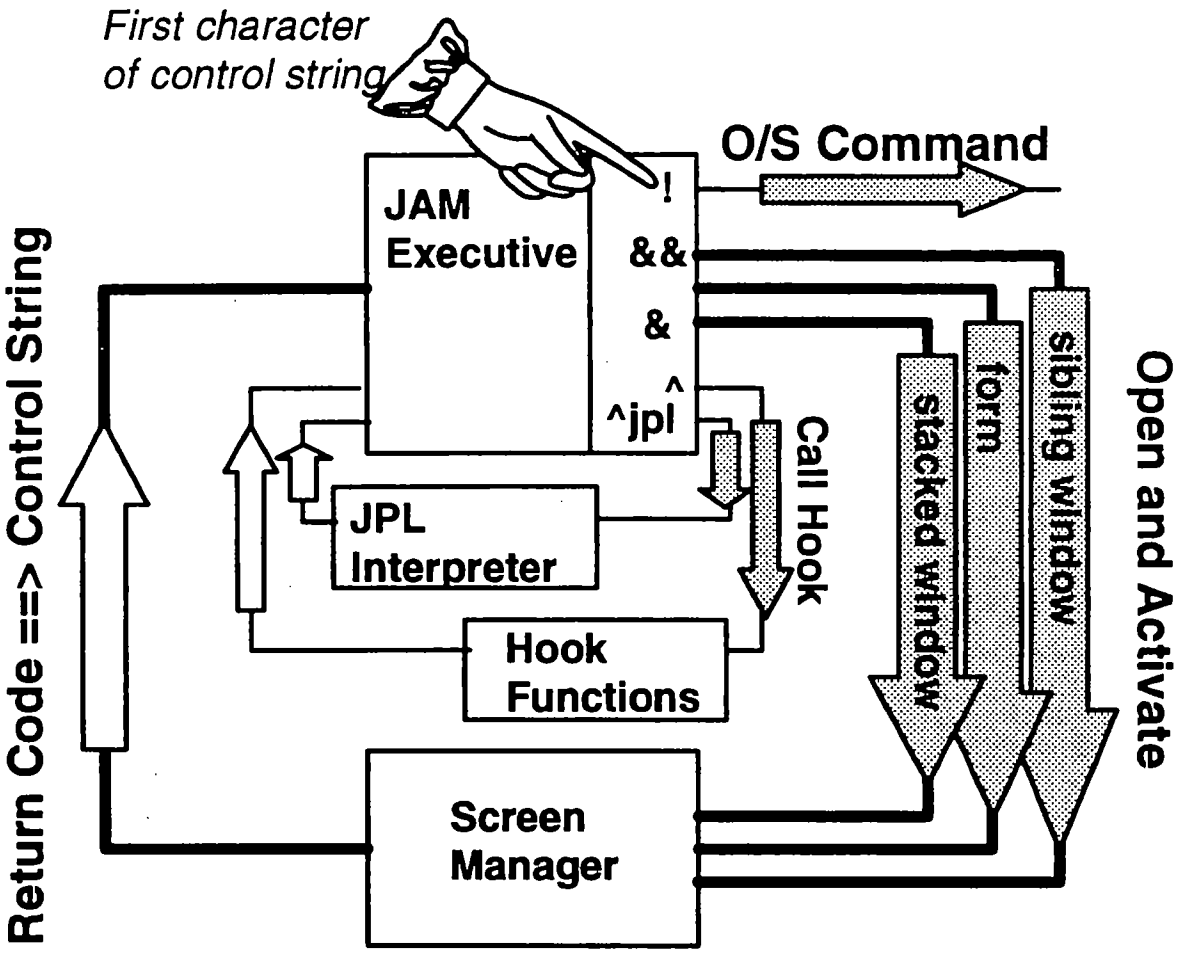


Figure 12: **JAM** Application Control Flow.

Note that in every case, the **JAM** Executive ends its cycle of action with a call to the Screen Manager function `sm_input`. The `sm_input` routine always returns to the **JAM** Executive with a function key or menu selection, which leads in turn to the interpretation of a new control string. It is also important to note that the Screen Manager itself can, at certain specific times, invoke JPL or C routines. In particular, hook functions can be attached to screen entry and exit, field entry and exit, group entry and exit, and field validation.

4.4.1

JAM's Dialogue in the Sample Application

In the case of our sample application, the JAM Application Executable is passed the name of the top level screen, `mainscrn`, as an argument. The JAM Executive, in turn, is invoked with that argument and asks the Screen Manager to open `mainscrn` as a form. The Screen Manager then handles the dialogue with the user, who enters a user name and password and strikes PF1. The Screen Manager returns to the JAM Executive, indicating that PF1 was struck, and then the JAM Executive examines the screen to find the associated control string. Since the control string `^jpl passcheck` is associated with the PF1 key, the JAM Executive invokes the JPL interpreter to execute the JPL `passcheck` procedure. `passcheck` checks the username and password combination, modifies the help text on the status line, and then toggles the screen to menu mode. It then returns to the JAM Executive, which instructs the Screen Manager to re–open the dialogue with the user, but at this time in menu mode.

The user selects the `personnel` menu choice, which causes the Screen Manager to return control back to the JAM Executive. The JAM Executive looks into the screen and finds that the control string `empscrn` is associated with the menu selection. The JAM Executive detects that the control string is requesting that control be transferred to a new screen, and directs the Screen Manager to open `empscrn` as a form and to start a dialogue with the user. On opening the screen, the Screen Manager invokes the screen entry jpl procedure, `authorize`. When `authorize` ensures that the user is authorized to use the sub–application, it returns to the Screen Manager. The Screen Manager then allows the user to tab through the form and enter information in the fields. If authorization fails, the JPL procedure forces return to the main menu.

When the user strikes the PF2 key to initiate a search, the Screen Manager returns to the JAM Executive. The JAM Executive finds the control string `^search` associated with the PF2 key, and invokes the developer–written C function named `search`. The function uses the screen field data entered by the user to structure a query and search the personnel data base. The response to the query is loaded into fields on the screen and control is returned to the JAM Executive, which directs the Screen Manager to re–open a dialogue with the user on that screen.

When the user strikes PF3 to update the employee, the Screen Manager returns to the JAM Executive which invokes the `update` C function to update the data base. On return from `update`, the JAM Executive directs the Screen Manager to re–start a dialogue on the `empscrn` screen.

The user strikes one of two keys, EXIT or PF10 to return from `empscrn` to `mainscrn`. The EXIT key has no associated control string, so the JAM Executive takes the default action of closing the current screen (`empscrn`) and re–opening the previously displayed screen. The PF10 key does have an associated control string,

namely the string "mainscrn". This directs the JAM Executive to close all open screens and to bring up mainscrn as a form. The two keys, in this case, bring the same result through two different mechanisms.

While on the empscrn screen, the user can strike PF1 to see the employee's salary history. The JAM Executive determines that the control string &salhist is associated with PF1, and directs the Screen Manager to display and activate the salhist screen as a window. When the salhist screen is activated, its screen entry function getsalhist is invoked. getsalhist is passed the identification number of the employee from the LDB, and the function fills the fields on the salhist screen with the results of a salary history query to the database.

While on the salhist screen, the user can scroll through the salary history records, but has no way of modifying them since they are protected from data entry. When the user strikes PF10, the mainscrn screen is displayed as a form, which causes both salhist and empscrn to be closed. The user can also strike the EXIT key. This causes the JAM Executive to direct the Screen Manager to close the current screen (salhist in this case), and re–activate the underlying empscrn screen.

4.5
KEEPING TRACK OF FORMS AND WINDOWS

As we discussed earlier, the EXIT key has a default action on a JAM screen. If the developer of an application does not associate a control string with the EXIT key, it causes the current screen to close and activates the previously displayed screen. If the active screen is a window, the window closes and the underlying screen is restored on the display and re–activated. If the current screen is a form, the form is closed and the previously visited *form*, not any intervening windows that might have been visited, is freshly opened. JAM's mechanisms for keeping track of previously displayed forms and of a stack of windows is discussed in the following sections.

4.5.1
The Screen Manager's Window Stack

The Screen Manager handles the overlaying of one window upon another. Internally, it maintains a stack of windows. The base of the window stack is always the currently displayed form. Any number of windows can be stacked one upon another, with the active window on the top of the stack. The Screen Manager handles the ordering of the

windows and the underlying data structures. The Screen Manager function `sm_r_window` displays a window over the existing screen, storing the obscured data in the process. The Screen Manager function `sm_close_window` closes the active window and restores the underlying data.

The **JAM** Executive uses `sm_r_window` when processing control strings starting with ampersands (`&` or `&&`). It uses `sm_close_window` to process the EXIT key when no control string is associated with EXIT and the active screen is a window. The **JAM** Executive does not maintain information about the ordering of windows in an application — that is entirely handled by the Screen Manager.

Note that the Screen Manager keeps all the information about all the windows in the window stack in memory. When a window is made active by virtue of an overlaid window being closed, it is not re–opened and re–displayed from the screen binary. Rather, the window is restored to its former state with information from the window stack.

The Screen Manager stacks windows one upon another as the **JAM** Executive processes successive control strings that start with ampersands. Control strings that invoke hook functions or operating system commands do not affect the window stack[12], but when a control string specifying form display is processed, all windows are closed and the window stack is purged from memory.

The same screen can appear more than once in the window stack. For this reason, developers should be aware that certain design strategies can use large amounts of memory. If `window1`'s PF2 control string is `&window2`, and `window2`'s PF1 control string is `&window1`, then successive striking of PF1 and PF2 will continue to pop up new instantiations of `window1` and `window2` continually allocating new memory on the window stack.

4.5.2
The JAM Executive's Form Stack

The **JAM** Executive maintains its own stack of screens, but only the screens that have been previously displayed as forms. The form stack is used to process the EXIT key thus allowing the end–user to return to an earlier portion of the application. The display of forms with the `sm_r_form` routine does not overlay existing screens, but closes them and clears the window stack. The `sm_r_form` routine, as part of the Screen

12. Unless the hook function explicitly calls library routines to display or close screens.

Manager, does not manipulate the form stack. The JAM Executive manipulates the form stack in conjunction with `sm_r_form` calls[13].

When the active screen is a form, and the user strikes the EXIT key, the default action taken by the JAM Executive is to remove the screen's name from the form stack and to direct the Screen Manager to display the previously visited form in the application. Successive strikes of the EXIT key will continue traversal of the previous forms in a Last–In–First–Out (LIFO) manner. When EXIT is struck on the top level form of the application, the application will terminate. The JAM Executive implements the ability to backtrack through forms by maintaining a form stack. Unlike the window stack, the form stack contains only screen names and is not used for the preservation of screen data. Only certain state information like cursor position and whether the form was being used in data entry or menu mode is preserved. This implies that changes made on a form and not explicitly saved by your application will not be automatically recreated when the form is re–visited[14].

The form stack is, in fact, not a purely LIFO structure. As the JAM Executive processes successive control strings requesting the display of forms, the names of the displayed forms are pushed onto the form stack. However, the JAM Executive searches back through the form stack before it pushes a new form name, and if the name to be pushed is found earlier in the stack, all intervening form names are popped off and discarded.

As an example, picture an application with three forms: `form1`, `form2`, and `form3`. `Form1` is the top level form, and is consequently the first form pushed onto the stack. The user then moves to `form2` by pressing the PF2 key, and `form2` is pushed onto the stack. Finally, the user presses the PF3 key. This displays `form3`, and pushes it onto the stack. At this point, the user is viewing `form3` as a base form, and `form1`, `form2`, and `form3` are on the form stack in that order.

On `form3`, the PF1 key has the control string `form1` associated with it. When the user strikes PF1, the `form1` screen is displayed again, but since it exists earlier in the form stack, the intervening references to `form2` and `form3` are discarded. When the user

13. Since the JAM Executive maintains the form stack, calls to Screen Manager form display routines in the sm_r_form family from hook functions may have unpredictable behavior. This is because the form stack will not be properly synchronized with the screens that were displayed previously in the application. In general, developers will want to use the supplied JAM Executive support routines sm_jform, sm_jwindow, and sm_jclose to open and close screens. These library functions, only available to those using the JAM Executive, ensure that both the form and window stacks are properly managed at all times.

The JAM Executive routine sm_jform calls the Screen Manager routine sm_r_form and appropriately manipulates the form stack. The JAM Executive routine sm_jwindow calls the Screen Manager routine sm_r_window. The JAM Executive routine sm_jclose calls the Screen Manager routine sm_close_window when the active screen is a window; but if the active screen is a form it pops the form stack and calls the Screen Manager routine sm_r_form to display the previously visited form.

14. Note that any changes made to fields that have corresponding entries in the Local Data Block will appear to be restored, but this is a feature of the Local Data Block processing and not JAM's form and window stacks. For more information on Local Data Block processing, please see section 4.6 on page 39.

strikes EXIT from the `form1` form, the application terminates since the last form in the form stack is popped[15].

This modified stack behavior for forms means that any given screen can occupy at most one slot in the form stack at any given point in time.

4.5.3
Stack Overview

In conclusion, the following window and form stack rules apply:

- When a screen is displayed as a form, the current window stack is closed and discarded.
- When a screen is displayed as a form, the form stack is searched for earlier instances of that screen. If it is found, all intervening screen names are popped off the form stack and discarded. If it is not found, the screen name is pushed onto the form stack.
- When a form is closed by the default action of the EXIT key, the form stack is popped and the previously displayed screen at the top of the form stack is re–displayed, without any regard to the previous state of the form. Any intervening windows are ignored. If the form stack is empty, the application terminates.
- When a screen is displayed as a window, it is laid over all currently displayed screens and placed on the window stack.
- When a window is closed by the default action of the EXIT key, the underlying screen is restored to the state it was in when it was de–activated. It is not re–opened.

4.5.4
The Form and Window Stacks in the Sample Application

In the sample personnel application, the form and window stacks never get very large. At the top level of the application, `mainscrn` is the only screen on the form stack. From `mainscrn`, when the user strikes the EXIT key, `mainscrn` is popped from the form stack and the form stack becomes empty. The application terminates.

15. By default, **JAM** will ask the user for confirmation before exiting the top level screen. A developer can also suppress default EXIT processing on that screen to make the application harder or impossible to leave.

When the user selects the personnel application and moves to the employee query screen, the name of the screen, `empscrn`, is pushed onto the form stack. From that screen, the user can use either EXIT or PF10 to return to the top level screen. If the user strikes EXIT, `empscrn` is popped from the form stack, leaving `mainscrn` on the stack. When the user strikes PF10, the modified push algorithm, explained in the preceding section, scans the form stack for a `mainscrn` entry. When found, all intervening screens (in this particular case, none) are popped and `mainscrn` is displayed.

When the user displays the salary history screen as a window, the form stack is not modified. `salhist` is pushed onto the window stack. When EXIT is pressed, `salhist` is closed, popped off of the window stack, and the underlying `empscrn` screen is restored and activated. When PF10 is struck, the **JAM** Executive first calls `sm_r_form` to close all windows and clear the window stack, and then searches for `mainscrn` on the form stack. When it is found, the intervening `empscrn` screen is popped off the stack and `mainscrn` is displayed again.

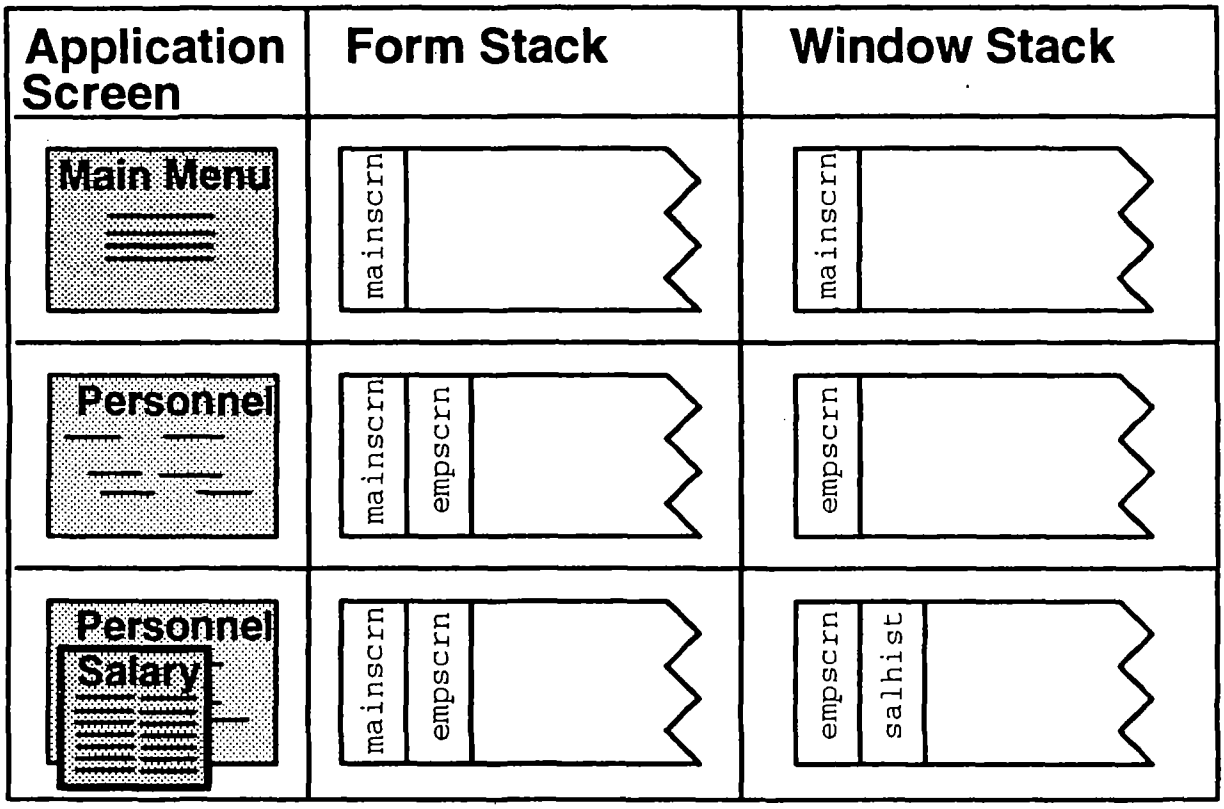


Figure 13: Form and Window Stacks in the Personnel Application. Note that the form and window stacks uniquely determine where you are in the application.

In Figure 13, the window and form stacks are pictured for every screen configuration in the application. Note that the behavior of **JAM** with regard to the form and window stacks implies the following points:

- No screen will be found more than once on the form stack.
- The application's top screen is always at the base of the form stack.
- The screen at the top of the form stack (the currently displayed form) is always at the base of the window stack.

4.6
LOCAL DATA BLOCK PROCESSING

As discussed earlier, the Local Data Block is a region in memory set aside for the storage of field data used by the entire application. It is dynamically created from the description of field elements in the data dictionary file for your application (section 2.3.2, page 11). Storage for LDB entry values is not allocated until it is used.

The LDB consists of entries indexed by name. These entries are directly correlated with any fields of the same name that exist on screens in the application[16]. Whenever a screen is opened or activated, any named fields that have corresponding entries in the LDB are initialized with data from the LDB. Whenever a screen is made inactive or closed, all the data in named fields that have corresponding entries in the LDB are written to those entries in the LDB.

The correspondence between field names and LDB entries allows **JAM** to provide the following features:

- Field data is automatically preserved as the user moves from screen to screen, provided that the fields in questions are linked to entries in the LDB.
- Application code uses the same **JAM** library routines to access fields on the active screen by name and to access LDB entries by name. This

16. The way a field is labelled on a screen has no relationship to its name. Field names are assigned to fields as identifying characteristics. A field might, for example, be labelled "First Name" but it might be named "fname". The field name, and not its label, is the identifying characteristic that relates it entry in the Local Data Block.

means that application code need not know whether a data item exists on the current screen. The JAM library calls `sm_n_getfield` and `sm_n_putfield` (and variants) will automatically access the screen if the named field exists on the current screen, or the LDB if the named field does not exist on the current screen.

In the sample human resources application, the user's username and password is saved in the LDB when the user moves from `mainscrn` to any sub–application. This means:

- Returning to the main menu does not require the user to re–enter a username or password.
- The screen entry routine `authorize`, used by every sub–application to ensure that the user is authorized for that sub–application, can access the username and password from the LDB.

In addition, in the personnel sub–application, the employee's name and identification number are stored in the LDB. This allows the employee's name to be automatically propagated from `empscrn` to `salhist`, and allows the screen entry function `getsalhist` on the `salhist` screen to access the identification number.

LDB entries can be initialized with data when the application starts by using initialization files. The syntax of these text files is described in the *Author's Guide*. The identification of these files to the application is described in the *Configuration Guide*.

4.7
JAM FLOW SUMMARY

In short, the program that drives a JAM Application, the JAM Application Executable, has a main loop that cycles back and forth between the JAM Executive, which processes control strings, and the Screen Manager, which displays screens and interacts with the user. The application architecture is summarized in Figure 14. The control strings are resident on screens.

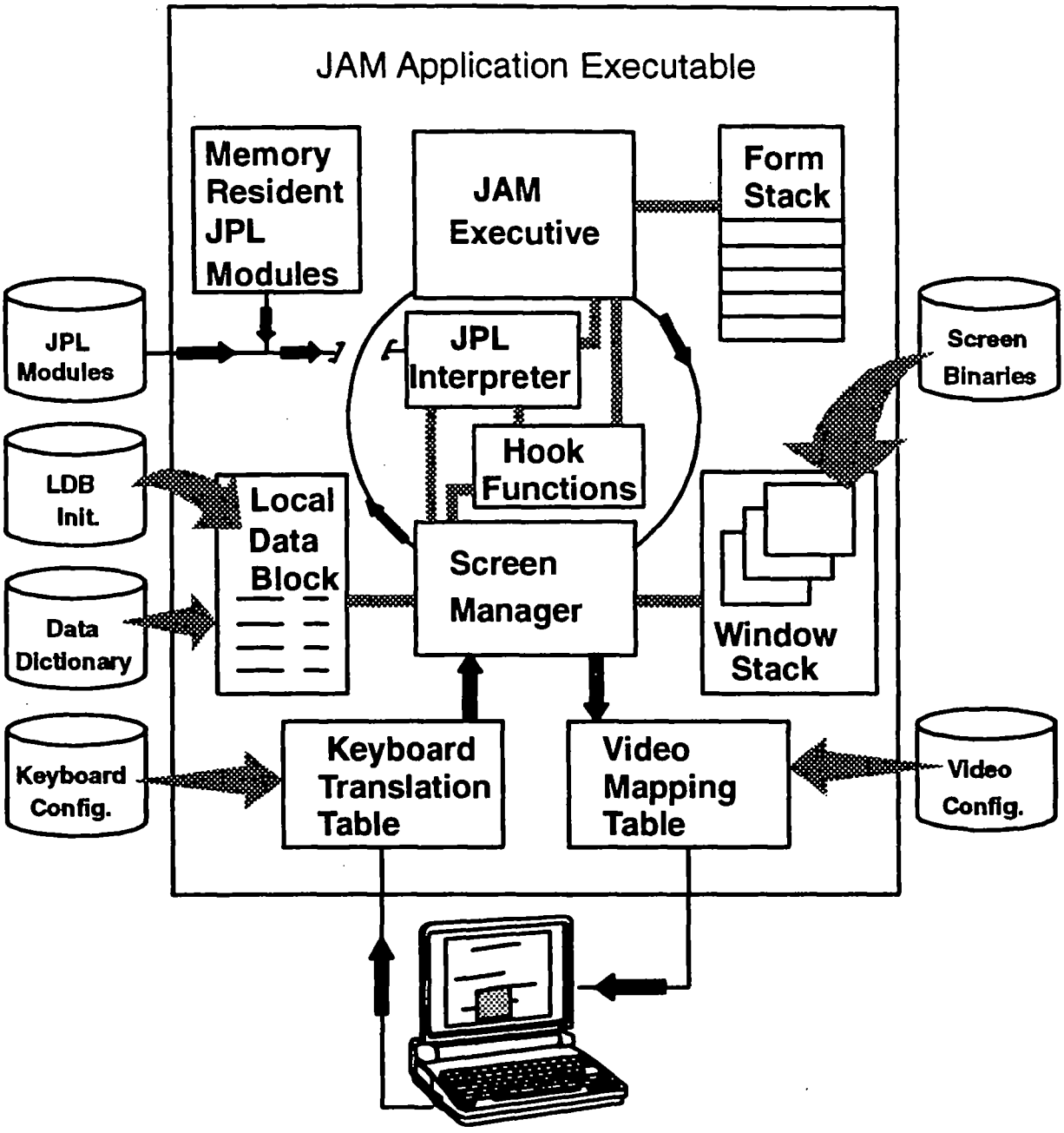


Figure 14: **JAM** Flow Summary

The Screen Manager maintains a window stack of all screens that are currently open. Only the top screen on the window stack is active, meaning that the user interacts with the application through that screen. The **JAM** Executive maintains a form stack of all the screens that are visited as forms in the application.

Through control string processing, the JAM Executive calls developer–written hook functions or JPL routines. The Screen Manager can call these same functions or routines at screen entry or exit, field entry or exit, group entry or exit, or field validation. Hook functions and JPL routines can invoke one another. Hook functions are written in C or some other third generation language and linked in to the JAM Application Executable. JPL routines are available to the JAM Application Executable as memory–resident data structures, as disk–resident files, or in screens.

The Local Data Block, a region in memory that assists in the sharing of field data between screens in the application, is created from the data dictionary when the application starts. It is also populated with data from its initialization files at that time. The Screen Manager stores screen field data into the LDB whenever a screen is made inactive, and restores screen field data from the LDB whenever a screen is made active.

To preserve display terminal independence, the Screen Manager does not read the user's input directly nor write to the user's monitor directly. User input is filtered first through a key translation table, so that any physical keystroke or sequence on the physical keyboard can be mapped to JAM's set of logical function and data keys. All application output is filtered through a video mapping table. The keyboard translation table is created at startup time from the keyboard configuration file for the terminal in use. The video mapping table is similarly created from a video description file for the terminal in use.

Chapter 5
JAM *Philosophy*

In this chapter, we briefly address various aspects of **JAM** and **JAM** development from a conceptual perspective. We discuss the ways **JAM** differs from other products you might use to create screen–intensive windowed applications.

5.1
JAM FEATURES

JAM has a number of features that make it a powerful development tool. In this section, we discuss some of these features, referencing other parts of the documentation where additional details can be found.

5.1.1
Display Hardware Portability

All **JAM** terminal output is filtered through a video mapping table. This table is created, when the application starts, from a video configuration file that contains escape sequences that the terminal in use understands.

The video configuration files are maintained as ASCII text and edited with any text editor. Their syntax is described in the *Utilities Guide*, along with the description of the `vid2bin` utility. The `vid2bin` utility takes the video configuration text files as input and creates binary video configuration files suitable for use by any **JAM** application.

JAM generally requires that the terminal in use be identified in the environment. For details, see the *Configuration Guide*.

5.1.2

Terminal Keyboard Portability

All JAM keyboard input is filtered through a keyboard translation table. This table is created, when the application starts, from a keyboard configuration file that translates physical keystrokes to JAM logical keys. These files can be used to support the delivery of an application to users of different types of keyboards, or by users to customize their own keyboards.

The keyboard configuration files are created either with a text editor or with the `modkey` utility. The `key2bin` utility takes these ASCII configuration files as input and creates binary keyboard description files suitable for use by any JAM application.

For details, see the *Utilities Guide* and the *Configuration Guide*.

5.1.3

Application Portability

JAM applications are generally portable across a wide variety of hardware platforms and operating systems. These range from personal computers running MS–DOS to mini–computers and workstations running UNIX to VMS super–minis. For details, see the *Configuration Guide*.

5.1.4

Data–Driven Soft User Interface

Many screen manager products generate source code that displays screens when compiled and linked into the application. JAM screens are created as binary data files and are manipulated by the Screen Manager as data structures. This means that screens can, at the discretion of those responsible for maintaining an application, be substantially changed after the application is on line without requiring re–compilation or re–linking. In fact, different users of the same application can have a different view into that application by virtue of using different screens.

Utilities are provided to generate source data structures from screen binaries for developers who wish to have screens compiled into the application. See the `bin2c` utility in the *Utilities Guide* for details.

5.1.5
Event–Driven Algorithm

Both the Screen Manager and the **JAM** Executive implement event driven algorithms. Developers write code in JPL or C that is executed by the Screen Manager and the **JAM** Executive when events occur. **JAM** handles the low level details of events, placing only minimal requirements on developer–written code. The Screen Manager handles events such as screen entry and exit, field entry and exit, group entry and exit, field and group validation. The **JAM** Executive handles events such as menu or function key selections. All information associating events with actions is stored in screens.

For more information on **JAM** Application Control Flow, see chapter 4 on page 17.

5.2
JAM DEVELOPMENT METHODOLOGY

There are at least three possible strategies for using **JAM** to develop and implement an application:

- Use the **JAM** Executive to prototype and/or ultimately implement the application.
- Use the **JAM** Executive to prototype the application, and then write an executive for the production version.
- Write an executive, as part of the bottom up development and implementation of the application.

The **JAM** Executive is a powerful routine which may appear deceptively simple. There are compelling reasons to use it in all but the rarest of cases. The strongest of these reasons follow:

- Use of the **JAM** Executive will allow modification and testing of the application during development under control of the authoring tool, `jxform`.
- Use of the **JAM** Executive will minimize the amount of time spent in compile–link–test cycles.
- Use of the **JAM** Executive will make maintenance of the application simpler and cleaner. Substantial modifications can be made to the user interface without changing the application code. In fact, the user interface may be modified in some cases without bringing the application off–line.

- Use of the **JAM** Executive will keep the application architecturally in line with the emerging dominant style of event–driven bit–mapped windowing applications.

There are few good reasons for not using the **JAM** Executive. These include:

- Some aspect of the **JAM** Executive algorithm is fundamentally incompatible with the application.
- The application has memory constraints that can not be met using the **JAM** Executive. This would typically happen only on operating systems that do not support virtual memory models.

For tips on writing an executive, please see the *Programmer's Guide*. Since we strongly recommend use of the **JAM** Executive, the remainder of the discussion that follows will assume its use.

5.2.1

The Use of Prototypes

In our consulting practice at JYACC, we emphasize application prototyping. **JAM** is especially well suited to development methodologies that rely heavily on prototyping since the absence of code does not restrict the application from traversing through the screens. This allows the development team to get a good sense of the look and feel of the final application, even before the hook functions are complete.

Projects undertaken with more traditional tools often lose momentum before completion because little of the progress can be noted early and changes in the specification lead to lengthy delays. An application shell of just screens, fleshed out gradually with more and more functionality in the code that runs behind the screens, allows a development project to have tangible success early on. In addition, the ease with which screens can be modified and manipulated during development *or after deployment* will bring timely gratification to the end users as their project specifications change and mature.

5.2.2

Design Strategy

JAM allows you to design your application in a fluid and natural top–down fashion. Create the screens you know about, link them together, and see what you have. Add a bit of code here and there. Add some more screens. Add more code. Change some existing screens. When the application feels right and the end–users like it, it can be released, but it is still quite possible for you to modify it further from time to time.

JAM allows end–users themselves to be an intimate part of the development process. An end–user can initiate a project without full specifications by generating a number of screens and establishing their hierarchy basically outlining the flow of an application. What better specification could there be? The authoring tool creates and modifies screens and links in a way that allows fine tuning changes to be made later. All the "real" programming that remains to be done is to flesh out the linked set of screens with function calls, either to interpreted JPL procedures or to lower level language functions that are compiled in to the **JAM** Application Executable itself. Typically programmers will write those functions, but they are at least partially specified by where they are included in the screen hierarchy.

The application evolves in a natural and intuitive way, with the authors continuing to manipulate and fine–tune screens, global data, and application hierarchy and control flow, while the programmers can concentrate on the underlying code that will be related to user events. Like word processing tools for documents, **JAM** allows the creators of applications to split the work into the technical and non–technical, the drudgery and the creative, and the design and the implementation, in a way that is fundamentally natural, intuitive and flexible.

Apart from being efficient, this aspect of **JAM** development allows individuals in organizations to cross traditional role boundaries. Programmers, end users, strategic planners, managers, and interface designers can all share in the actual implementation of the product, which can lead to enhanced communication and a better application. This malleable aspect of **JAM** development has further reaching consequences as well. It can allow for degrees of informality in the development process that would lead to chaos in the traditional model. Distinctions between specification, prototyping, design, implementation, and piloting can be blurred, which can lead in turn to applications that more comfortably fit the tasks of the individuals who use them.

INDEX

Symbols

A

C

D

E

F

H

I

J

K

L

M

O

P

R

S

T

U

V

W

X

New Features in JAM Release 5

Chapter 1
Categories of New Features

This document summarizes the major new features and product enhancements of **JAM** Release 5. These features are organized into 8 groups:

- Screen and Window Management
- Keyboard and Input Management
- Widgets and Menus
- Field Edits and Attributes
- Screen Editor Enhancements
- Improvements to JPL
- Internationalization
- Miscellaneous

Chapter 2
Summary of New Features

2.1
SCREEN AND WINDOW MANAGEMENT

2.1.1
Viewports

The viewport facility enables the use of *virtual screens* that are larger than their display size. You may even create virtual screens that are larger than the physical display. The viewport facility determines which portion of a virtual screen is visible at a given time, as well as the size and position of the "viewport" into the virtual screen, that appears on the display.

As a user tabs through the fields in a virtual screen, **JAM** automatically scrolls the viewport, bringing the necessary fields into view at the appropriate time. The user may optionally use the VIEWPORT key to move, resize, and scroll the viewport manually. This key also controls sibling windows, discussed later in this section.

To pop up a window called "xyz" at line 5, column 10 in a 10 by 40 viewport, the following control string would be specified:

```
&(5,10,10,40)xyz
```

The last two parameters are optional. If omitted, the viewport and screen sizes will be the same, unless the virtual screen is larger than the display. In that case, the viewport will be the size of the display.

If the virtual screen has a border, scroll bars appear in the right and bottom borders indicating that only part of the screen is visible. The size of the scroll bar indicates what

percentage of the screen is shown, while its position indicates which section of the screen is visible.

2.1.2
Shrink–to–Fit Windows Option

Developers may now make a window *shrink–to–fit*. This function call will dynamically shrink a window based on the amount of data in the screen. The "shrink–to–fit" feature is particularly useful in item selection lists when the developer doesn't know the number of items that will be present. For example, in a scrolling array with 15 on–screen elements where only 3 lines are retrieved, it appears awkward in a "normal" window because of the 12 blank lines. A "shrink–to–fit" specification would reduce this window by 12 lines. This feature is enabled by a call to the function `sm_shrink_to_fit`.

2.1.3
Relative Placement of Windows

In Release 4, windows are specified by the actual line and column at which they are to appear on the display. Release 5 allows window positions to be specified in relation to each other.

For example, assume three automatic windows are brought up next to each other on line 10 of the screen. If they are specified with absolute coordinates and later need to be modified to appear on line 9, each window must be modified. When relative coordinates are used, only the position of the first window needs to be changed. A plus (+) or minus (–) sign is used to indicate relative placement. In the example below, the window will open 5 rows *below* and 3 columns to the *right* of the top left hand corner of the calling screen:

```
&(+5,-3)windowname
```

2.1.4
Help Screens with JAM Control Strings

Release 5 supports JAM control strings on help and item selection screens. This enhancement enables paging through item selection lists that are stored in external files. Additionally, globally defined function keys will now work on help screens.

2.1.5

Sibling Windows

In **JAM** 4, windows are stacked with only the topmost window accessible to the user. Programs select underlying windows via the wselect library routine. With **JAM 5**, developers can invoke a window and designate it as a sibling . Siblings are considered to be at the same level as the current window, as opposed to stacked on top of it. The user can switch freely among sibling windows via the viewport key.

There may be multiple levels of sibling windows. For example, if a sibling window is active and a stacked window is opened, the underlying sibling windows cannot be selected by the user until the stacked window is closed. Stacked windows, however, may open other sibling windows which can be freely selected by the user.

Sibling windows are specified in **JAM** control strings by two ampersands (`&&`*`windowname`*). Stacked windows are specified by a single ampersand.

Sibling windows are handled by a single call to a new input routine (defined below). Return occurs when a function key is pressed or a menu selection is made, but not when the user moves from one sibling window to the next. If necessary, the program can make calls to see which sibling was active when an event occurred, or the program can process the windows in a pre–determined order, whichever is more convenient.

2.1.6

Screen Entry and Exit Routine Enhancements

In version 4, screen entry functions were called when a screen was opened; exit functions were called when the screen was closed (i.e. removed from the window stack). Release 5 provides an additional option to call these functions when the screen is made active and inactive. The *why called* parameter tells the routine the circumstances under which the routine was called.

A screen is made active when it is first opened or when a window on top of it is closed. A screen is made inactive when a window on top of it opens or when the screen is removed from the window stack. The active/inactive hooks are especially useful for setting screen–wide options such as installing keychange functions.

JAM 5 resolves the problem in screen entry procedures which prevented access to screen fields that exist in the LDB. This eliminates the current work–around solution of calling `sm_allget`. The similar problem of setting fields in screen exit procedures has been eliminated as well. This is discussed further in the *Programmer's Guide*.

2.2 KEYBOARD AND INPUT MANAGEMENT

2.2.1 New Keyboard Input Routine

Release 5 provides a new keyboard input routine that handles menus, data entry screens and hybrids. This replaces the `sm_menuproc`, `sm_openkeybd` and `sm_choice` functions of Release 4. The old routines will still be available in Release 5 for compatibility.

Screens with *only* data entry fields will operate in data entry mode; screens with *only* menu fields will operate in menu mode. For screens with *both* types of fields, a developer may specify which mode to start in. A menu toggle key has been introduced to allow switching between the two modes. This routine makes the `jam_menu` field obsolete.

2.2.2 Mouse Support

JAM responds to mouse clicks in a number of ways:

- During data entry, clicking the mouse cursor on a field will reposition the text cursor to that field.
- Clicking while the mouse cursor is positioned on a menu or radio button field will select that item.
- Clicking on a checklist field will toggle its state.
- If the status line contains mapped keytops (%K...), clicking on a keytop is equivalent to typing that key.
- If keysets are used, clicking on the soft key label is equivalent to typing that soft key.

2.2.3 Key Remapping

In JAM 5, any cursor or edit keys can be mapped to any other key without calling the key change functions. For example, the NL (Return) can be easily changed to XMIT using a screen entry function.

2.2.4

Developer–Specified Backward Tabbing Order

Using the PREVFLD field edit, developers can now define the order in which backward tabbing will occur.

2.2.5

Soft Keys

Certain terminals on the market, notably those from Hewlett Packard and AT&T, have areas on the screen in which you can display the actions associated with the various function keys. In Release 4 we provided library routines for writing text into these labels, but left the application responsible for ensuring that those actions occurred. In Release 5 we extend this capability by introducing the concept of soft keys.

Soft keys are keys whose logical value may change during the course of an application. In one context, a particular key may translate to HELP, while in another context, the same key may act as PF1. The associated screen labels indicate what action a soft key will take. This feature is especially useful on terminals with a limited number of function keys. The logical translations and screen label text for soft keys are defined in a *keyset*.

In the key translation file for a terminal, the function keys, or some other set of keystrokes, are defined as the soft keys. When one of these is pressed, **JAM** uses the current keyset to translate the keystroke into a logical value. In the absence of a keyset the translation is to the corresponding PF key (i.e. — *soft key1 = PF1*, etc.).

Release 5 provides a Keyset Editor which allows keysets to be built and maintained. Keysets may contain multiple rows of key definitions. Each definition contains a label, an optional display attribute, and the translated logical value. A MORE key may be defined that allows the user to view the next row of labels.

Several levels of keysets are provided. There can be an *application* level keyset as well as *screen* level keysets. When in the Screen Editor there is an additional *system* level keyset, used for **JAM** navigation and the Screen/Data Dictionary Editors.

JAM must be configured properly in order to enable soft keys, and an entry must be made in the video file. Soft keys may be simulated on terminals for which there is no hardware support.

2.3 WIDGETS AND MENUS

2.3.1 Radio Buttons

radio buttons present a list of choices to the user, but allow one, and only one, to be selected. The choices are displayed on the screen as in the example below:

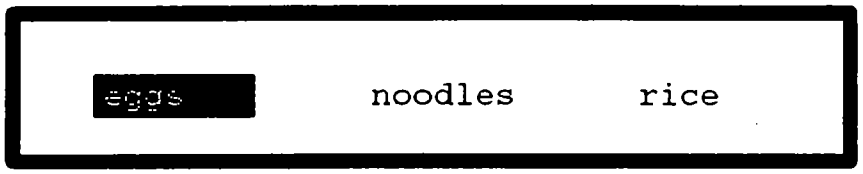


Figure 1: A radio button grouping.

The text in the choice fields is protected from data entry. The user makes a selection by positioning the bounce bar on the desired choice, and pressing the select key, as with menus.

2.3.2 Checklists

Checklists are similar to radio buttons except that any number of entries, including *none*, may be selected. Checklists come in three styles:

- *Button* lists work similarly to a menu, in that you move a bounce bar to the desired option and press a select key.
- *Check box* lists contain fields to the left of the choices that are marked when selections are made. The character(s) used to mark check boxes are stored in the video file, so they can be changed from terminal to terminal.
- *Check box with bounce bar* is similar to check box, except that as the cursor moves through the boxes, a bounce bar tracks the cursor over the choice fields to the right of the boxes.

There may be multiple checklists and radio buttons on the same screen. Checklists and radio buttons are treated as data entry fields for the purpose of the menu toggle capability.

2.3.3

Field Groups

Checklists and radio buttons are called *field groups*, because they are usually composed of several distinct fields. For the purpose of tabbing however, each group is treated as a single field. That is, a tab will cause the cursor to leave one group and go to another group or field. As such, there are group entry and exit functions, and next and previous group edits.

Field groups must have names that are distinct from field names and the names of other groups. A program may access a group by name, in which case it will get the entries that have been selected. Groups can also appear in structures and in the LDB, thereby permitting the passage of selected entries from one screen to another.

2.3.4

Menu Selection

The user may select from menus and checklists by:

1. pressing the NL key (usually ENTER).
2. pressing the XMIT key.
3. typing the first uppercase letter of a menu item.

2.3.5

SAA Compliance

The Release 5 input routine provides an option to leave sub–menus open after a selection is made. This brings **JAM** pull–down menus into compliance with SAA/CUA standards.

2.4

FIELD AND EDIT ATTRIBUTE FEATURES

2.4.1

Currency Fields

Release 5.0 allows considerable flexibility in specifying numeric and currency fields. The following criteria may be specified:

- Decimal symbol
- Number of decimal places (min and max)
- Thousands separator
- Currency symbol (up to 5 characters)
- Placement of currency symbol (left, right, decimal point)
- Local format (developer defined mnemonic that defines each of the above)
- Rounding (floor, ceiling, round)
- Justification (left, right)
- Fill character
- Unlimited number of currency formats in the same screen

2.4.2
Numeric Fields

Developers can specify the default radix separator (decimal point) in the message file.

2.4.3
Date and Time Fields

In addition to the ability to specify almost any date and time format, developers may now define mixed time and date formats in a single field. Ten pre–defined local formats are provided.

2.4.4
Extended Display Attributes

Release 5.0 adds support, both for new display attributes, and new combinations of attributes. Among these are:

- Independent foreground and background colors
- Background highlighting
- Graphics attribute

- Standout attribute

The graphics attribute is used on terminals which implement line drawing graphics characters with on–screen or area attributes (such as Hewlett Packard terminals). Release 4 was limited to support for latch–type attributes for graphics.

Standout can be implemented at the user's discretion. For example, it might be used for italics on terminals which support such a font.

Release 5 supports the notion of a *working pen* in the Screen Editor. With this feature, display data is created using a certain type of pen. If you are using a HIGHLIGHT RED pen for instance, new characters will be shown in highlight red. When you change the pen, new characters are written with the new colors and attributes. You can change pens every time you select new attributes, via the PF4 key. An option on this window allows you to change the current display area, the pen or both. Pressing PF4 with the cursor outside of a field or display area has the effect of only changing the pen. This permits display text with different attributes to appear adjacent to each other in screens.

2.4.5

Support for Null Values in Fields

When a field is created, the screen designer can specify what the field should look like if it contains no data. Whenever the field is empty, this null value string will be shown in the field. Library functions are available to determine if a field contains a null (eg. `— sm_is_null )`

2.4.6

Changes to Field Naming Conventions

Release 5 relaxes the restrictions on field names, so that, among other things, field names may contain characters in foreign alphabets.

2.4.7

Field Entry and Exit Routine Enhancements

A new type of field edit, *field exit function*, has been added. This is primarily used to undo anything that may have been done in the field entry procedure. Currently, the field validation function must be used to do this, but in the default case, field validation routines are only called when the user tabs or returns out of the field.

JAM guarantees that screen and field entry and exit functions will be called symmetrically: If an entry function is called, the associated exit function will be called exactly one time. Furthermore, if the exit function is called and the screen or field is revisited, the entry function is guaranteed to be called again. In summary, if you need certain options to be set for a given screen or field, you can set them in the entry function and reset them in the exit function.

The developer may install default screen, field, and group functions to avoid attaching the same edit on all fields when the function to process them is the same.

2.4.8
Error Message when a JPL or C Procedure is not found

In Release 5, JPL or C procedures which cannot be found at runtime will cause an error message to be displayed.

2.4.9
Extended Data Types

Packed and zoned decimal types are provided for use in data structures.

2.5
SCREEN EDITOR FEATURES AND ENHANCEMENTS

2.5.1
Block Move and Copy

Multiple fields may be moved and copied in groups while maintaining their relationship to each other on the screen.

2.5.2
Clipboards

Release 5 introduces a Screen Editor feature called *clipboards*. Clipboards are available for moving data between screens, or to temporarily save fields which must be deleted to make room for other fields. 26 clipboards are available.

Groups of fields can be saved in clipboard "template" files and later recalled and merged into new screens. This is useful for creating objects (e.g. name and address blocks) and populating them on multiple screens.

2.5.3
Line Drawing

Release 5.0 supports a line drawing mode, in which the arrow keys can be used to draw lines in any one of ten different styles. As you change directions, **JAM** provides an appropriate corner character. If you cross another line of the same style and attribute, an intersection character is generated. To draw a box, one may specify the corners and **JAM** will create the connecting lines of the box all at once. Line styles can be specified in the video file.

2.5.4
Minor Screen Editor Improvements

Some of the major improvements have been cited in other sections. In addition we have the following enhancements:

- The Screen Editor always begins in DRAW mode
- **JAM/DB***i* processing is turned off in the Screen Editor to avoid lengthy time–outs and to prevent developers from inadvertently putting default data into fields
- In DRAW mode, the tab key stops at all fields and display areas.
- `sm_install` may specify that a user function should be turned off in (ie. — not called by) the editor.

2.6
IMPROVEMENTS TO JPL

2.6.1
JPL Tokenization

JPL procedures can be *tokenized* and placed in binary files. This substantially speeds up runtime parsing. This is also called *JPL compilation.*

2.6.2

JPL Libraries

JPL procedures can now be stored in *form libraries* or within an *executable*. This enhancement eliminates the need to keep many files around at runtime. It also eliminates the possibility of users inadvertently modifying JPL procedures.

2.6.3

JPL PROCedure

Multiple JPL subroutines (or procedures) can be declared and stored in the same edit or file. These procedures can be made available only to other procedures in the same file or edit, or to all procedures. Named JPL procedures can be attached to screens via the PF3 window of the Screen Editor. They are not automatically executed when the screen is opened, but are available to field–level JPL as subroutines, and to function keys which invoke JPL procedures via `^jpl`. In addition, an *unnamed* procedure attatched to a screen is executed when the screen is opened. This allows for the creation and initialization of variables that are global to the procedures attatched to the screen.

2.6.4

JAM Library Access from JPL

Access to many of the **JAM** library routines is provided via JPL. Parameters are parsed and converted into the types needed by the called functions. This capability is also available for developer–written functions. It works as follows:

When a function is specified in a function list, the function name may be followed by a *function prototype*, which defines the number of parameters the function needs and their types. **JAM** picks this up when the function is called and converts the arguments appropriately.

2.7 INTERNATIONALIZATION

2.7.1 Creating Applications for Non–English Speakers

JAM can now be used to develop applications for use with any language that is read from left to right and that uses characters that can be represented in eight bits of information. While it has always been possible to create "non–English" applications, some of JAM's features had to be disabled to do so. Release 5 makes it possible to use JAM's full capabilities when creating such applications, and makes it easier to convert applications from one language to another and to write applications for use with more than one language.

Text appearing in JAM applications generally comes from screens and the message file. To convert an application from one language to another, the display data on all screens and the text in the message file must be modified. The length of text strings may be changed freely without affecting program operation.

This works well for prompts and messages, but JAM must also accommodate the various conventions for displaying and entering dates, numeric data and currency information. JAM 5 handles this by allowing the screen designer to specify Currency and Date/Time formats for fields. This can be done on a field by field basis, or globally, so the formats need be specified only once for the entire application. If specified globally, the formats are stored in the message file so they may be changed along with message text to accommodate different languages or local customs.

An application may support more than one language, by having a set of screens and messages for each language. The appropriate language set is then selected at initialization, either through an environment variable or user prompt.

2.7.2 Localizing JAM for Non–English Speaking Developers

Assuming appropriate licenses have been granted, the JAM package itself can be converted to languages other than English without modifying source code. The localiza-

tion process involves changing the **JAM** screens (including help screens), translating the documentation, and modifying the message file. Once this is done, all prompts, messages and status text is shown in the new language. Even mnemonics for specifying date, time and currency edits will be translated. Programmers, though, still must invoke library routines by their existing English–based names.

2.8
MISCELLANEOUS

2.8.1
Utility to Convert Screens Between ASCII and Binary

A new utility, `f2asc` is available for converting screens created with the Screen Editor to an ASCII format suitable for use with a text editor. The ASCII file can subsequently be converted back into a **JAM**–compatible form.

This feature can be useful for keeping screens under a source code control system such as SCCS, copying JPL procedures from one screen to another, or making global changes to screens via a text editor.

2.8.2
User–defined Synchronized Arrays

Release 4 includes rules for determining when arrays are considered synchronized, they must:

1. start on the same line
2. have the same number of on–screen occurrences
3. have the same number of total occurrences
4. have the same distance between elements

In Release 5, fields that do not meet these criteria can also be designated as synchronized, as long as they have the same number of on–screen and off–screen of occurrences. This permits multi–line scrolling arrays, which are particularly useful in database applications.

2.8.3

Function Naming Convention Changes

Functions attached to fields and screens may have names that contain up to 31 characters.

2.8.4

Disk–based Scrolling

Release 4 provided the ability to define scrolling fields with a maximum of 9,999 occurrences each. Data in these fields was stored in memory (allocated as needed as entries were made). Nonetheless, if all occurrences were filled, memory for all entries had to be allocated.

In Release 5 we provide the ability to write scroll drivers and use these drivers to handle off–screen data. Each driver is given a name which may be attached to scrolling fields when they are created. When these fields scroll, **JAM** calls entry points in the appropriate driver to get and put off–screen occurrences.

Drivers may be written to access sequential text files, database tables or even the LDB. This allows the application to use less memory and perform operations such as scrolling backwards through database tables, which are not possible with standard scrolling.

Sample drivers are provided by JYACC along with documentation for writing custom ones. Refer to the section on Alternative Scrolling in the **JAM** *Programmer's Guide*.

2.8.5

Performance Improvements

Release 5 supports various performance optimizations, targeted toward exploiting terminal capabilities in order to reduce screen output. Algorithms for area and on–screen attributes have been refined as well.

The screen stack has been reorganized. In Release 4, **JAM** had a single screen area for the current display. When a window was opened, underlying data would be copied to a save area, and then restored when the window was closed. In Release 5, each window has its own save area which gets mapped to the physical screen when the screen is updated. This dramatically accelerates moving and `wselect`–ing windows. It also makes opening and closing windows somewhat faster.

2.8.6

New Library Routines

Over 65 new library routines have been added. Groups of new routines are:

- Routines to access to all global variables
- A function to truncate an array and all synchronized arrays
- Functions to get and put **JAM** control strings
- A function to keep a screen image in memory. All screen data, including field contents, scroll buffers and relevant structures stay intact when the screen is closed. When the screen is subsequently reopened, the structures in memory are reused rather than recreated. See `sm_svscreen` in the **JAM** *Programmers Guide*.
- A function to set a rectangle on the screen to a given background attribute. See `sm_bkrect` in the **JAM** *Programmers Guide*.
- Functions to associate developer–defined pointers with screens, so that one can maintain screen–specific data without having to create and maintain a custom window stack.

2.8.7

Block Mode Terminal Support

Release 5 of **JAM** provides support for block mode terminals such as the IBM 3270, and other terminals that do not interact with the host computer on a keystroke by keystroke basis. All of **JAM**'s help and windowing capabilities are available.

While any screens created with the Screen Editor will work with block mode terminals, care should be taken when using certain features to avoid placing a data entry burden on the user. For example, shifting fields will not automatically shift when the cursor gets to the end of the field. **JAM** will permit the user to use the shift left and shift right function keys or the zoom key.

Other things to consider:

- field entry and exit functions are disabled during block mode.
- field validation functions are called during screen validation only.
- character edits are deferred until screen validation, unless handled by the terminal.
- screen, data dictionary, and Keyset Editors will not work in block mode due to their highly interactive nature.

- normally, terminal characteristics are specified in video files. This is not possible with block mode because of the vast differences between different block mode terminals. Therefore, for each type of block mode terminal, a set of "driver" routines must be written. JYACC will provide sample drivers and documentation for writing custom ones.

Author's Guide

TABLE OF CONTENTS

Chapter 1
Introduction

1.1
SHOULD YOU READ THIS GUIDE?

You should read this guide if you will be authoring applications with **JAM**. Since authoring is the theme that unifies **JAM**, you should also read this guide, perhaps at a faster pace, if you are working with **JAM** in other ways. In either case, first read the Overview; it discusses terminology and concepts that are essential to understanding this guide.

JAM is designed to help you, the application builder, rapidly build sophisticated applications. You use **JAM** to provide the look and feel required by the users, and to provide application–specific processing. **JAM** helps you by taking the drudgery out of:

- creating complex screens,
- interconnecting screens,
- linking screens with database operations, and
- linking screens with customized application logic.

The process of creating, interconnecting, and linking is called *authoring*. Other processes are required to build applications with **JAM**, including the processes of configuring **JAM**, adding application–specific processing using procedural languages such as JPL (JYACC's Procedural Language) and C, and adding application–specific database processing using **JAM/DBi** (JYACC's **JAM** Database Interface). The table below helps you determine which **JAM** guides contain detailed information relevant to your task.

If you are doing:	*Then you will find details in:*
Authoring	**JAM** *Author's Guide*
JAM Configuration	**JAM** *Configuration Guide*
JPL Processing	**JAM** *JPL Guide*
C Processing	**JAM** *Programming Guide*
Database Processing	**JAM/DBi** *Guide* for your database

In all cases, please read the **JAM** *Overview* first.

Chapter 2
Keyboard Entry

2.1
INTRODUCTION

The purpose of this chapter is to explain keystroke translation and data entry conventions.

JAM applications are developed for an imaginary keyboard, called the *logical keyboard*, that has more keys than any commercially available *physical keyboard*. The logical keyboard includes keys such as `PF1`, `DELETE CHAR`, `ZOOM`, `TRANSMIT`, `EXIT`, `HELP`, `A`, and `X`. A **JAM** key translation file defines, for a particular physical keyboard, the translation from physical keystrokes to logical keys. For example, the IBM PC keyboard has no physical key labelled `ZOOM`. The IBM PC key translation file distributed with **JAM** defines the physical keystroke `Alt z` to be the logical `ZOOM` key. This physical key to logical key translation ensures that any **JAM** application will work with almost any physical keyboard.

In the **JAM** documentation, unless otherwise explicitly noted, a reference to a key is a reference to a logical key. For example, "Press the `HELP` key" means "Press the combination of physical keystrokes that is translated to the logical `HELP` key by your key translation file". To use **JAM**, you must document (at least mentally) the key translation that you are using. You must also know what actions **JAM** will take as you press logical keys. The purpose of this chapter is to assist you in both areas. If you wish to change your key translation, or wish to create a key translation, please read the *Utilities Guide* description of the `modkey` utility. Key translation occurs in the **JAM** library function `sm_getkey`, and may be changed by the user-definable function `sm_keychg`. These functions are described in the *Programmer's Guide*.

Your key translation is defined by your key translation file. The *Configuration Guide* contains a complete description of how to configure your software environment in order to use a particular key translation file. You may need to read and understand your key translation file in order to document the key translation you are using. The `modkey` utility, described in the *Utilities Guide*, is helpful in interpreting key translation files.

2.2
DOCUMENTING YOUR KEY TRANSLATION

JAM comes with over 300 pre–defined logical keys, 255 of which are the logical data keys like `p`, and `#`. JAM translates the physical data keys on your keyboard into the corresponding logical data keys, without using the key translation file. For example, the physical `p` key is always translated into the logical `p` key. Logical data keys are displayed in whatever manner your video monitor normally displays physical keys. Therefore, for portability, only data keys corresponding to ASCII display characters should be used[1].

The JAM logical keyboard is shown in Figure 1 on page 6. Note that for readability, data keys (`Q`, `W`, `E`, `R`, `T`, `Y`, ...) and keys that are rarely used (`PF13-24`, `SPF13-24`, `APP13-24`, and `SFT1-24`[2]) are omitted from the diagram. We suggest that you use this keyboard diagram to document the physical keystrokes corresponding to each logical key shown. In Figure 2 on page 7, we have done this for the IBM PC keyboard, according to the key translation file provided with JAM for that keyboard.

JAM provides a mechanism, called keytops, for displaying the physical keystrokes corresponding to a logical function key for the keyboard currently in use. For example, the

1. The ASCII character set is composed of eight bit characters in the range of 0 to 255 (hex FF). Characters in the ranges hex 20 to hex 7E and hex A0 to hex FE are ASCII data characters. Characters outside of those ranges are ASCII control characters. Control characters have mnemonic names; the character hex 1B (decimal 27) is called the escape key, or ESC. When you press a physical key, the keyboard generates a sequence of one or more characters. JAM converts these characters into logical keys by assigning a logical key number between 1 and 65535. Logical values between 1 and hex FF represent logical data keys and are displayable data. Logical values above hex FF represent the other logical keys. ASCII data characters received from the keyboard are assigned the logical key number equal to their ASCII value (i.e., they are not really translated). Sequences beginning with an ASCII control character are translated, via the translation defined by the key translation file, to a logical data or function key. If an ASCII control character does not begin any sequence of physical keys defined in the key translation file, then it becomes a logical data key; this is useful for machines, such as IBM PCs, that use ASCII control codes for displayable data (although it is inherently non–portable).

2. Soft keys are most commonly used with terminals that have physical soft keys. Therefore, we have chosen not to clutter the logical keyboard diagram with soft keys

keytop for the logical EXIT key on an IBM PC keyboard is usually `Esc`, since the `Esc` key is usually the physical key corresponding to EXIT. Keytops are described in the *Configuration Guide*, in the `modkey` section of the *Utilities Guide*, in the Status Text section of this guide (page 56—there is an example as well), and in the status line function sections of the *Programmer's Guide*.

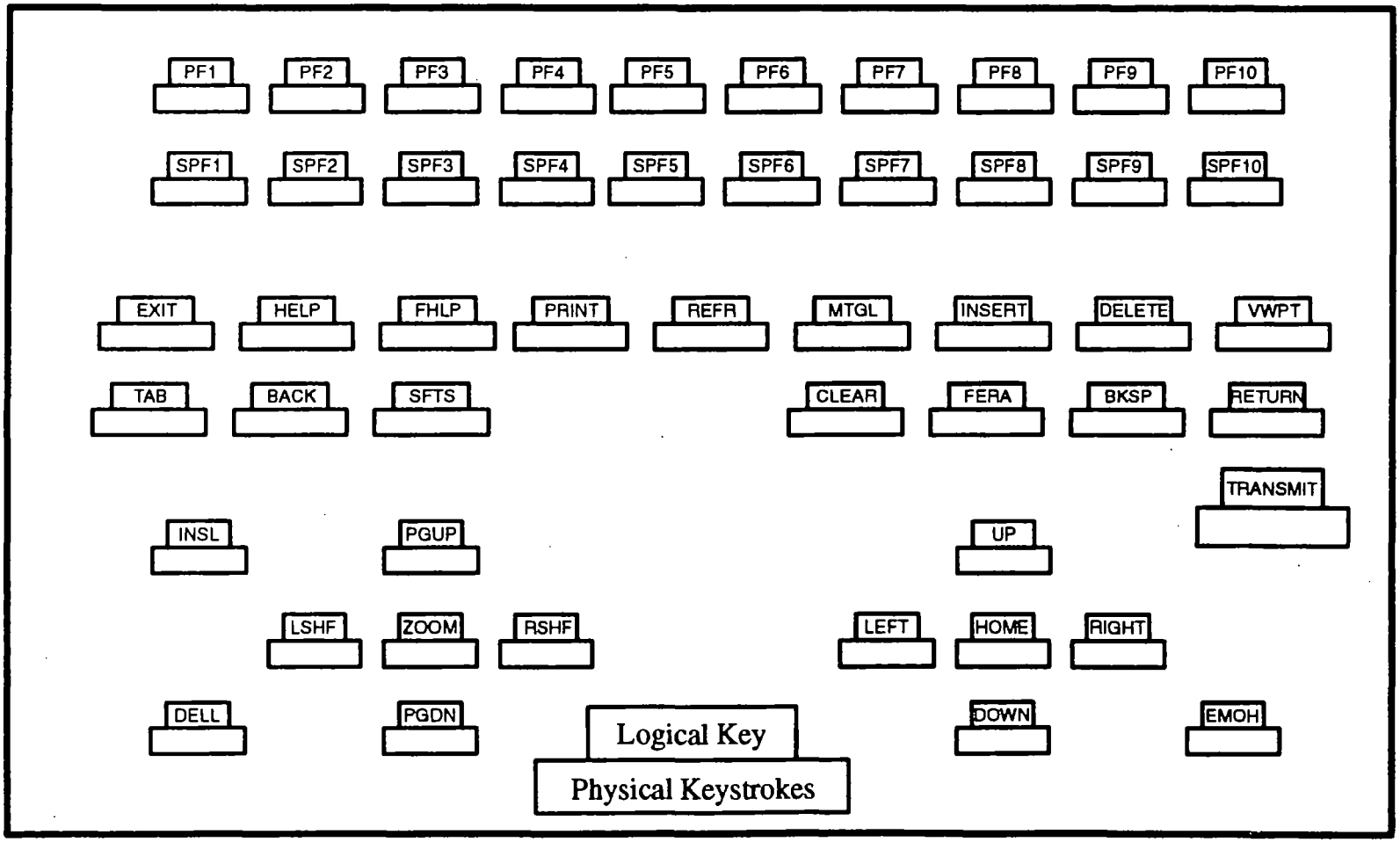


Figure 1: **JAM Logical Keyboard Template**

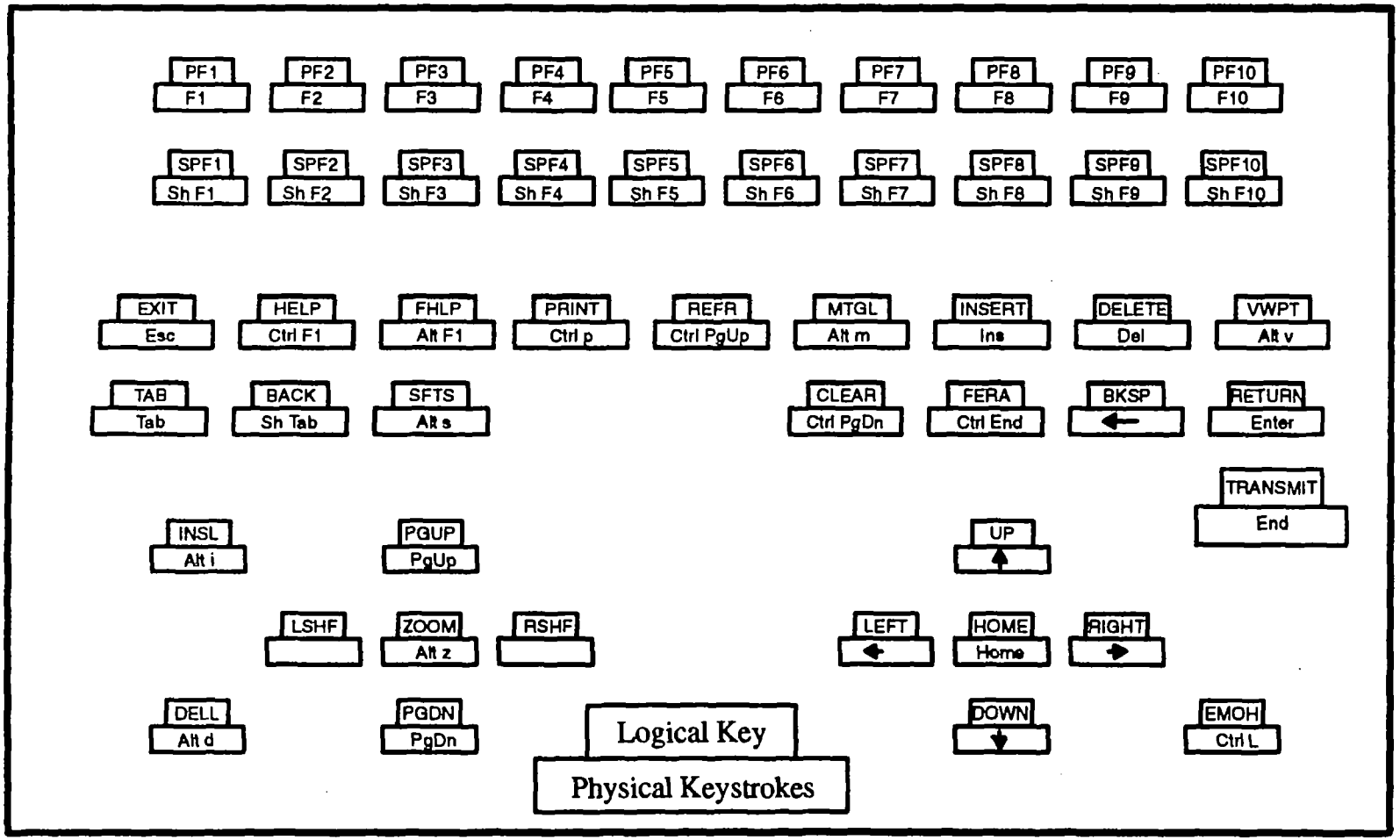


Figure 2: **JAM Logical Keyboard Template for the IBM PC**

2.3
LOGICAL KEY ACTIONS IN GENERAL

The action associated with a logical key may vary between applications, screens, fields, or modes as defined by the application developer. For example, on a menu, the space bar moves between menu selections; on an unprotected field, the space bar leaves a space — and these default behaviors may be modified by the developer (see the *Configuration Guide*). In this section, we list all of the translated **JAM** logical keys, and the actions usually performed by each key. The behavior of keys in specific situations, such as within menus, item selection lists (see page 54), checklists and radio buttons (see section 4.3.10), is discussed in this section, as well as in the corresponding sections of the Authoring Reference chapter. Note that the behavior of keys in these situations may be drastically changed by a developer. See the *Configuration Guide* and the **JAM** library function `sm_option` for details.

In the **JAM** Logical Key Translation table below, Short Name is the logical function key name used when programming. Long Name is the logical function key name commonly used when describing the logical key. The physical keystroke column is left blank for you to document your physical to logical key translation.

Figure 3: **JAM** Logical Function Keys

Short Name	*Long Name*	*Description*	*Physical Keystrokes*
APP1–APP24	APP1–APP24	Application function keys. Typically associated with application specific actions.	
BACK	BACKTAB	Move cursor to previous field. Validation normally inhibited.	
BKSP	BACKSPACE	Delete character to left of cursor and move the cursor to that position.	
CLR	CLEAR ALL	Clear all clear–unprotected occurrences on active screen. System date and time fields are updated.	
DARR	DOWN ARROW	Validation normally inhibited. In a scrolling array, scroll through the allocated occurrences of this array before moving on to another field.	
DELE	DELETE CHAR	Delete character under cursor.	
DELL	DELETE LINE	Delete the data in the current line, and move up all data in occurrences below this one in this and all synchronized arrays not protected from clearing. In draw mode, delete the current line and move up lines below.	
EMOH	LAST FIELD	Move cursor to beginning of last tab–unprotected field (reverse of HOME).	
EXIT	EXIT	Return to the next higher level screen.	
FERA	FIELD ERASE	In left–justified field, clear from the cursor to end of field. In right–justified field, erase entire field. System date and time fields are updated.	
FHLP	FORM HELP	Display help window for screen.	
HELP	HELP	Display help window for field, or for screen if field has no help.	
HOME	HOME	Move cursor to first tab–unprotected field. If no field is tab–unprotected, move to top left–hand corner of screen.	

Short Name	*Long Name*	*Description*	*Physical Keystrokes*
INS	INSERT CHAR	Toggle mode of data entry between insert and overwrite. Initial mode is overwrite.	
INSL	INSERT LINE	Insert a blank line, moving down all data in occurrences below this one in this and all synchronized arrays that are not protected from clearing. Fails if last occurrence in any array is already filled. In draw mode, move down the current line and lines below.	
LARR	LEFT ARROW	Validation normally inhibited. At beginning of shifting field, shift field and synchronized fields to right.	
LP	LOCAL PRINT	Print the screen image or save it to a file. See SMLPRINT in the *Configuration Guide.*	
LSHF	LEFT SHIFT	Shift field to the left.	
MTGL	MENU TOGGLE	Toggle between data entry and menu modes. A mouse click will also perform this action.	
NL (newline)	RETURN	Move cursor to first tab–unprotected field below current line, wrapping to the top line if none are below. If in last element of a scrolling array, scroll down this and synchronized arrays. In draw mode, move cursor to beginning of next line.	
PF1–PF24	PF1–PF24	Program function keys. Typically associated with application specific actions.	
RARR	RIGHT ARROW	Validation normally inhibited. At end of shifting field, shift field and synchronized fields to left.	
REFR	RESCREEN	Clear physical display and re–draw screen.	
RSHF	RIGHT SHIFT	Shift field to the right.	
SFT1–SFT24	SFT1–SFT24	Soft keys. Soft keys are translated into other **JAM** logical function keys.	
SFTN	NEXT ROW	Select next row of soft keys.	None — SFTN can only be assigned to SFT1–SFT24

Short Name	*Long Name*	*Description*	*Physical Keystrokes*
SFTP	PREVIOUS ROW	Select previous row of soft keys.	None — SFTN can only be assigned to SFT1–SFT24
SFTS	SOFTKEY SELECT	Toggles between application and system (e.g. `jxform`) keysets.	
SPF1–SPF24	SPF1–SPF24	Shifted program function keys. Typically associated with application specific actions.	
SPGD	SCROLL UP	Scroll array and synchronized arrays up several lines. Scrolling up means moving towards the beginning of the array.	
SPGU	SCROLL DOWN	Scroll array and synchronized arrays down several lines. Scrolling down means moving towards the end of the array.	
TAB	TAB	Move cursor to next tab–unprotected field. If in last such field, move to first such field. Field validation is performed. Cursor remains in field if validation fails. In draw mode, move to next field, regardless of protection, or to next display data.	
UARR	UP ARROW	Validation normally inhibited. In a scrolling array, scroll through the allocated occurrences of this array before moving on to another field.	
VWPT	VIEWPORT	Enter viewport control mode. The viewport may be moved, re–sized, or shifted. Sibling windows may be activated.	
XMIT	TRANSMIT	Make changes in active screen effective, often closing active screen. All fields on screen are validated.	
ZOOM	ZOOM	Expand the scrolling and/or shifting field into a pop–up window. Data may be changed in the pop–up window.	

2.4
DATA ENTRY

At runtime, data entry is permitted only in fields that are not protected from data entry. When you type a data character, it is copied to the field under the cursor, subject to certain rules and restrictions. The rules and restrictions depend on the character and field edits applied to the field. A character edit rejects unacceptable characters as they are typed, usually with a bell. For example, no letters may be typed into a `digits only` field. A field edit rejects an unacceptable field as a whole, and repositions the cursor at the beginning of the rejected field[3]. A field edit is applied only when the field is validated.

Field validation normally occurs when a field is tabbed out of or is completely filled. In addition, all fields on the screen are validated when TRANSMIT is pressed. If an entry fails its field edit, then the cursor is repositioned at the beginning of the field. The field may be exited, without validation, by using the arrow keys. Note that the beginning of right–justified fields is the rightmost position of the field. As new characters are typed into the beginning of a right–justified field, the remaining characters shift to the left.

The behavior of **JAM** during data entry processing may be changed by the library function `sm_option`.

2.5
MENUS

In a menu, there is a reverse–video cursor, referred to as a *bounce bar*, that appears over the current menu choice. The TAB, RIGHT ARROW, DOWN ARROW, and space bar keys all move the cursor to the next choice. The BACKTAB, LEFT ARROW, UP ARROW, and BACKSPACE keys all move the cursor to the previous choice. Pressing TRANSMIT or RETURN selects the menu choice under the bounce bar. Alternatively, typing the first character of a choice (actually, enough characters to identify the choice uniquely) will cause that choice to be selected.

It is possible to have a screen behave alternatively as a data entry screen and as a menu screen. The MTGL (menu toggle) key toggles the active screen between data entry and menu mode. A mouse click will also perform this action. Clicking in a menu field toggles into menu mode. Clicking in a data entry field toggles into data entry mode.

3. A field edit may also specify justification and upper/lower case translation. These edits are applied as characters are typed, and are not part of the validation process.

The behavior of **JAM** during menu processing may be changed by the library function `sm_option`. For example, the selection can be based on the first upper case character of a choice rather than the first character.

2.6 ITEM SELECTION

An item selection screen provides a list of choices from which the user may choose in order to enter data into a field (see page 54). The effect of keystrokes on an item selection screen is identical to the effect of keystrokes on a menu screen.

The behavior of **JAM** during item selection processing may be changed by the library function `sm_option`.

2.7 CHECKLIST AND RADIO BUTTON GROUPS

The user can select one or more fields of a group (exactly one in the case of radio button groups). Therefore, **JAM** displays both the cursor position and the current selections from the group. The cursor may be represented by a normal cursor, by displaying the current field in reverse video (i.e. with a bounce bar), or by displaying the field with blinking (or some other attribute that can be seen in a field with a bounce bar). Each selected field may be represented by a bounce bar or by a checkbox to the left side of the field.

Each example below shows a group with three choices: eggs, noodles, and rice (also known as the selection text). In each example, eggs are selected, and the cursor is on noodles. The examples use reverse video and blinking attributes (blinking looks like this), but the actual attributes depend on the capabilities of the video display. Furthermore, the examples apply equally well to radio buttons and checklists.

Figure 4: Group With No Boxes

Figure 5: Group With Boxes and Bounce Bar

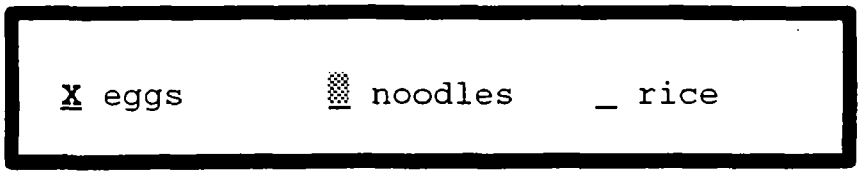


Figure 6: Group With Boxes and No Bounce Bar

The space bar moves the cursor to the next choice *within* the group. The TAB key moves the cursor out of the group. The UP ARROW, DOWN ARROW, BACKTAB, LEFT ARROW, RIGHT ARROW, and BACKSPACE keys work as they do in the absence of a group. Typing enough characters to identify a choice uniquely, selects that choice. Pressing RETURN (not TRANSMIT) selects the choice under the cursor. If the group contains radio buttons, then selection of a radio button automatically de–selects the previous selection.

The behavior of **JAM** during group processing may be changed by the library function `sm_option`.

Chapter 3
The Authoring Environment

In this chapter we present a description of operation of the authoring utility `jxform`. A developer generally uses `jxform` in the following ways:

- test the application in application mode. This is the top level mode of operation for `jxform`. In application mode, a developer can run the application, ensuring that the links from screen to screen are properly designed.
- enter the Screen Editor to create and edit screens. A developer can quickly switch between screen development and testing.
- enter the Data Dictionary Editor to create and edit the data dictionary. A developer maintains a list of data elements and their characteristics in the data dictionary to ensure that fields found on more than one screen have the same characteristics, and to ensure that the actual data are consistent from screen to screen at run time.

3.1
ENTERING AND EXITING THE AUTHORING UTILITY

The **JAM** authoring utility is entered by typing the following command at the operating system prompt:

```
jxform screenname
```

where ***screenname*** specifies the name of the top level screen for the application you wish to author. `jxform` searches for the file containing the screen. The name of the

file, usually ***screenname***.jam, depends on the environment and/or operating system in use. If ***screenname*** is omitted or is not found, it may be specified after jxform starts.

The display will clear, and the authoring utility will attempt to find a data dictionary and initialize the local data block (see the Glossary and the Overview for explanations of terms not defined here). You may see some diagnostic messages from the system, and a directive on the status line asking you to hit the space bar to continue. If you have not created a data dictionary, or if the data dictionary has errors, a message such as the following appears:

```
No Data Dictionary file.
Index not initialized.
```

These messages are informational. You can run **JAM** without a data dictionary.

If ***screenname***.jam exists, then it will display as a form with the status line shown below.

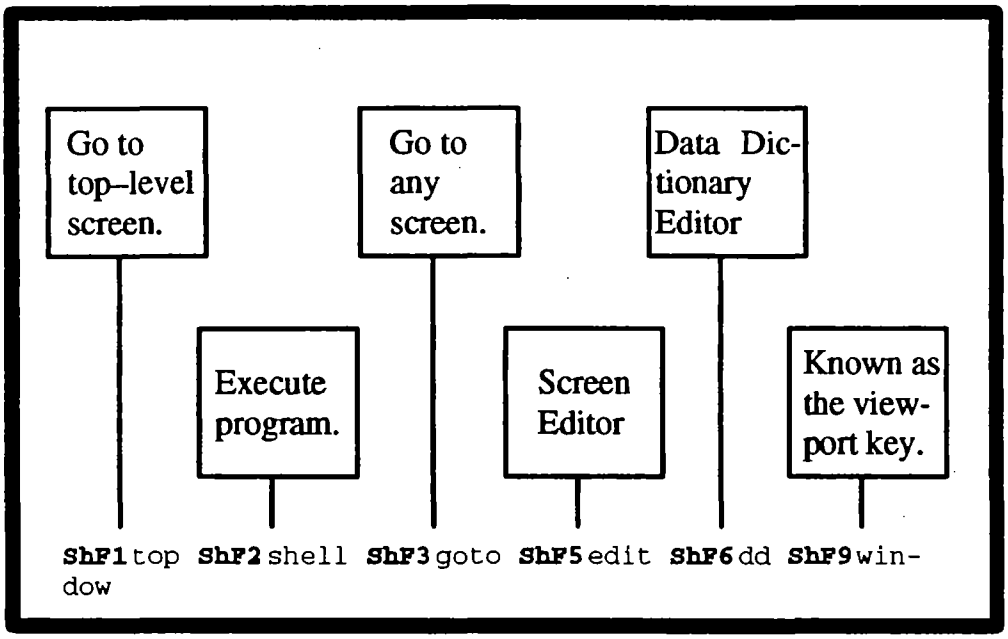


Figure 7: Status Line in Application Mode

To exit the authoring utility, press EXIT. You will be asked for confirmation before being returned to the operating system.

3.2
OPERATING IN APPLICATION MODE

When you enter the authoring environment as described above, you are placed in *Application Mode*. In this mode, you can simulate the application you are authoring. This

means that you can test all the following aspects of the application as if it were in runtime mode:

- Data entry with all edits applied.
- Special processing including help, item selection, and table lookup.
- JPL and C functions (other languages if you have a **JAM** language interface) associated with screen, field, and group entry/exit.
- Screen–to–screen movement.
- JPL and C functions associated with function keys and menu selections.
- Exits to other programs and to operating system commands.

To test C functions, you must create an executable program that contains those functions along with the rest of the authoring utility. The *Programmer's Guide* contains instructions for making your own authoring executable.

Certain function keys have special meaning in application mode and during runtime. In application mode, they are displayed on the status line as listed below. At runtime, they will not be displayed unless explicitly displayed by the author.

- `SPF1:` Go to top screen. The top level screen of the application is displayed. This clears all other screens from the form and window stacks.
- `SPF2:` Execute operating system command. You are prompted for an operating system command. **JAM** handles setting and resetting display terminal characteristics. When the operating system command terminates, you must press the space bar to continue.
- `SPF3:` Go to any screen. You are prompted for the name of the screen. This allows you to jump anywhere within an application.
- `SPF9:` Control the viewport of the active screen. You can move or resize the viewport, re–position the screen beneath the viewport, or switch to a sibling window. For more information on viewports and sibling windows, see the Authoring Reference chapter.

There are three additional special keys that are active only in application mode of the authoring utility.

- `SPF4:` Invoke the Keyset Editor. This key is active only if soft key support is included as explained in the *Configuration Guide*.
- `SPF5:` Invoke the Screen Editor.
- `SPF6:` Invoke the Data Dictionary Editor.

Chapter 4
The Screen Editor

4.1
INTRODUCTION

In this chapter we present a detailed description of the operation of the Screen Editor, presenting the features in the order that they are encountered by a developer. A developer generally uses the Screen Editor in the following ways:

- create and edit screens. A developer uses the Screen Editor to populate screens with fields, display data, and screen–to–screen links. Once screens are created, they are stored in files as screen binaries.
- compose the links to C functions and JPL procedures. The links are the bridges to application processing. A developer creates and edits the references on screens that cause transfer of control to C functions and JPL procedures.
- test the functionality of individual screens. Inter–screen testing, on the other hand, is done from within application mode of the authoring utility.

Most of the development work done while authoring is done in the Screen Editor. This includes placing graphics and text on the screen, populating the screen with fields, menus and groups, attaching hook functions written in C or JPL, and specifying links to other screens and programs.

As of release 5, the `f2asc` utility is provided to convert screens between the binary format used by the Screen Editor (and at runtime) and an ASCII text format. This feature can be used to make global changes (such as adding the prefix `gl_` to all field

names) with an operating system utility (e.g. a text editor or a find and replace program) that are more difficult through the Screen Editor. It can also be used to place screens into source code control systems (such as SCCS and RCS in UNIX). Please see the *Utilities Guide* for more information.

4.2
ENTERING AND EXITING THE SCREEN EDITOR (SPF5)

The Screen Editor can be entered from application mode by pressing SPF5. The Screen Editor can be entered directly from the operating system by typing:

```
jxform -e screenname
```

You may specify a list of screens to edit by typing:

```
jxform -e screenname1 screenname2 ...
```

Note that certain platforms support wildcard expansion for specifying multiple screens.

The display will clear, and the Screen Editor Entry Screen will be displayed as shown below. The screen has two fields; one for the name of the screen to edit and one, which is optional, for the name of a screen to use as a template. Screens are always created and edited as files[4].

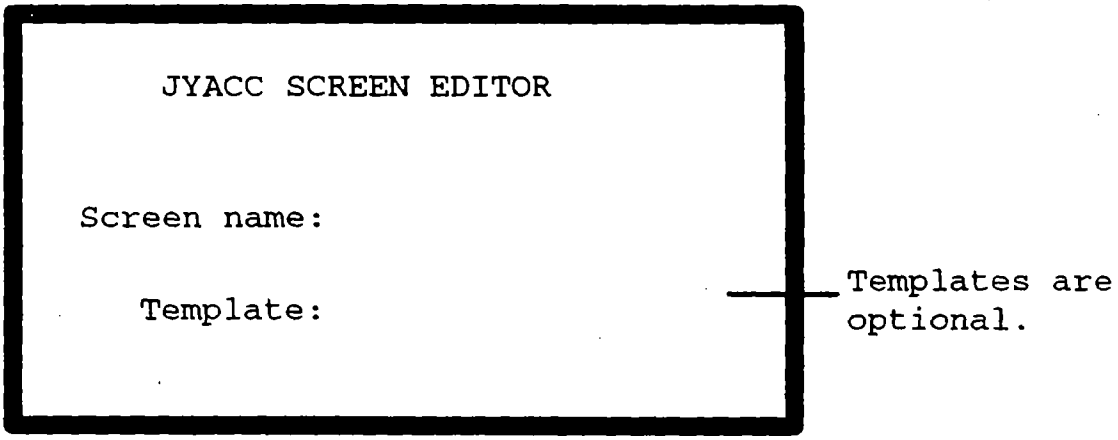


Figure 8; Screen Editor Entry Screen

4. JAM is packaged with utilities that convert screen files into C language source code data structures so that programmers can include memory–resident screens in their runtime versions of JAM. Screens may also be stored in screen libraries. It is important to note, however, that the Screen Editor can operate only on screen files.

A template screen provides a starting point for a new screen; templates help to standardize the screens in an application. *Caution: A template should not be used when editing an existing screen; the template will replace the existing screen.* If you specify a template screen, you will be asked to accept the template by pressing XMIT, or to reject it by pressing EXIT.

Once the Screen Editor Entry Screen is complete, press XMIT to begin editing the screen. If the screen you specify does not exist, then you will be creating a new screen. Initially, the Screen Editor will be in DRAW mode (the word DRAW appears on the status line). The status line shown below will be displayed (the status line is terminal dependent). It indicates which function keys can be pressed to move fields, assign field characteristics, etc.

```
F2draw F3scrn F4fld F5summ F6sel F7move F8copy F9rep F10more
```

Figure 9: Screen Editor Status Line

Pressing EXIT will exit the Screen Editor as long as no pop–up windows are open and the Screen Editor is in DRAW/TEST mode (as opposed to SELECT or LINE DRAW modes, discussed later). The Screen Editor Exit Screen, shown below, is displayed.

```
save screen xtmenu.jam
rename screen and save it under new name
continue processing this screen
process another screen
exit from Screen Editor
```

Figure 10: Screen Editor Exit Screen

The menu in the Screen Editor Exit Screen has the following options:

- `save screen` ***screen_filename***
  Save the screen on disk in a file named ***screen_filename***.
- `rename screen and save it under new name`
  Save the screen under a new name. You are prompted for the new name. The original file is untouched.
- `continue processing this screen`
  Return to the Screen Editor.
- `process another screen`
  Edit another screen. If a list of screens was specified (e.g. `jxform -e s1.jam s2.jam`), then the next screen in the list will be presented.

Otherwise you are prompted for the screen name. Warning: the current screen will not be saved.

- `exit`

  Return to application mode. Return to the operating system if the editor was invoked with `jxform -e`.

4.3
SCREEN EDITOR FUNCTIONS

The functions and modes of the Screen Editor are accessed via the function keys PF2 through PF10 and SPF1 through SPF9. PF2 through PF10 are displayed on the status line, while SPF1 through SPF9 are displayed on the Shifted Function Key Menu – which is accessed by pressing PF10. The ordering of this section follows the ordering of the function keys, as listed below.

- PF2 Draw/Test Mode Toggle.
- PF3 Screen Characteristics.
- PF4 Field Characteristics.
- PF5 Field Characteristics Summary.
- PF6 Select Mode. This enables use of the clipboard and block move/copy. The actions of the function keys changes in select mode.
- PF7 Move.
- PF8 Copy.
- PF9 Repeat Last Action.
- PF10 View Shifted Function Key Menu.
- SPF1 Assign Control Strings to Function Keys.
- SPF2 Create Special Objects. Shortcut methods for creating menus, radio buttons, checklists, and screen name fields.
- SPF3 Show Field and Group Names.
- SPF4 Data Dictionary Search
- SPF5 Add to Data Dictionary
- SPF6 Group Attributes. Create or change radio buttons and checklists.
- SPF7 Synchronize Arrays.

- SPF8 Special Graphics Characters.
- SPF9 Line Drawing Mode.

4.3.1

Draw and Test Modes (PF2)

The Screen Editor always begins in *draw mode*, the basic mode for creating and editing fields, display data, and borders. The status line contains the word DRAW. In draw mode, you can enter data anywhere on the screen; use the arrow, TAB, and NL keys to position the cursor. Fields can be created by typing successive underscores, and then pressing XMIT. The display shows an exact image of the edited screen, except that fields appear as underlined or highlighted areas, regardless of their display attributes.

From draw mode, you can enter *test mode* by pressing PF2. The PF2 key is a toggle between draw mode and test mode. The status line will change by replacing the word DRAW with the word TEST. Underscores will be converted to fields. Test mode is used to try out the screen, to see whether it operates as planned. All screen and field characteristics are in force, but it is not as complete as the simulation available in application mode because control strings cannot be executed. This means there is no way to test application flow from screen to screen in test mode[5].

In both draw mode and test modes, the EXIT key displays the Screen Editor Exit window. In either mode, you can press any of the function keys listed on the status line, PF2 through PF10. The shifted function keys SPF1 through SPF9 are also available, although their functions are different than those they have in application mode. The features accessible from function keys are described in the following sections.

Draw and test modes also function differently with respect to pressing HELP or SCREEN HELP. In draw mode, you receive context sensitive help for using the Screen Editor. In test mode, you receive the developer–defined context sensitive help that is attached to the field or the screen.

Display Data

Display data is constant data on the screen, such as screen headings, field labels, and graphics. Borders, status line messages, and data in fields are not considered to be display data. See pages 96 and 97 for a discussion of creating character graphics and drawing lines and boxes.

5. Hook functions (except those invoked from control strings) written in JPL are executed in test mode, but those written in the C language are invoked only if they are compiled into a customized authoring executable. See the *Programmer's Guide* for information about building a customized authoring executable to run the Authoring Utility.

To enter display data on the screen, the Screen Editor must be in draw mode. If the editor is in test mode you must press PF2 to get there. Use the arrow keys, the TAB key, and the NL key to position the cursor, and then enter characters via the keyboard. Any character can be entered. However, characters defined to be draw field symbols (often the underscore) will be converted into fields when XMIT is pressed.

We need to introduce the notion of a *display area*. This is an area of display data that is treated as a unit. It consists of display data that are on one line, not interrupted by a field, not spanning any change in display attributes or color, and not separated by more than one blank. Note that in draw mode, pressing the TAB key will move the cursor to the next display area, field, or border position.

As you enter display data on the screen, you have some basic editing capabilities. The BACKSPACE key deletes the character just before the cursor and moves the cursor back one space. The DELETE key deletes the character under the cursor, and the INSERT key toggles the character entry mode between insert mode and typeover mode. If the terminal supports more than one cursor style, then **JAM** will adjust the cursor to indicate insert/typeover mode. The FIELD ERASE key will erase characters to the end of a display area. The INSERT LINE and DELETE LINE keys insert and delete entire lines on the screen and shift the remaining lines appropriately, although they will not do anything if adding or deleting a line would change the distance between elements of an array (arrays are discussed on page 27). SELECT mode (see page 92) provides more advanced block move and copy capabilities.

Display Attributes

While you are entering display data, press PF4 to set display attributes and colors. This will cause the Display Attributes Screen to pop up, as shown below.

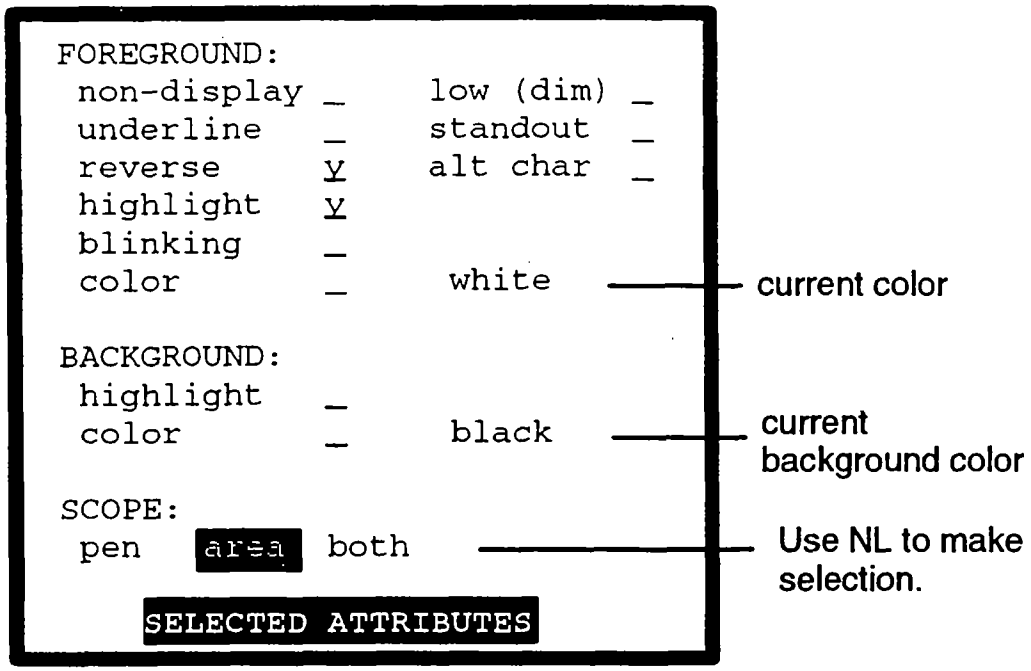


Figure 11: Display Attributes Screen

If the cursor is in a display area or field, then the attributes of that area or field are shown. The foreground and background attributes are set by typing `y` or `n` into each field to be modified. Note that most PCs do not support highlighted background colors, and many terminals do not support background colors at all. Typing `y` into either `color` field will cause the Color Menu to display as shown below (typing `y` is the only way to display the Color Menu).

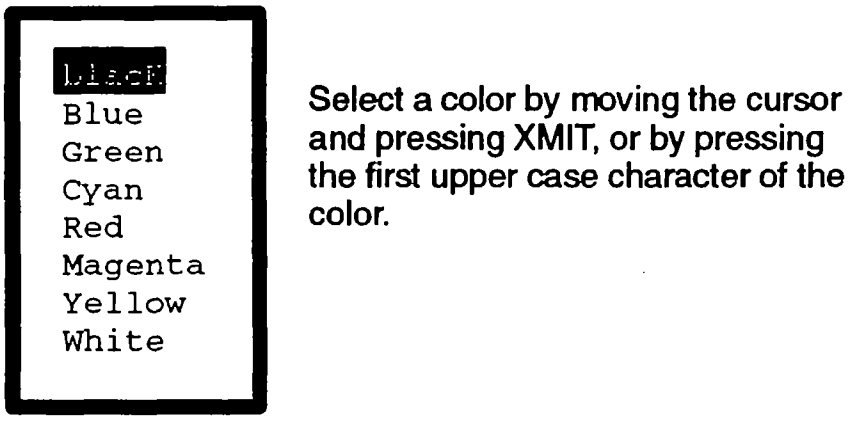


Figure 12: Color Menu

You can assign an attribute even on a terminal that does not support that attribute (e.g. color on a monochrome terminal). **JAM** will attempt to simulate the attribute (e.g. underlines can be simulated with underscores), or will ignore the attribute (as in the case

of color on a monochrome terminal). Note that this is true everywhere that display attributes can be specified, including fields, display areas, borders, and screen background.

The selected attributes are shown in context at the bottom of the Display Attributes Screen in the field marked `SELECTED ATTRIBUTES`. This way they can be seen before they are applied.

You have the choice of setting attributes for the display area beneath the cursor (i.e. the field or display text), for the pen, or for both. Make this choice in the `SCOPE` section of the Display Attributes Screen. `SCOPE` is a radio button (see page 85). Move the cursor to the `pen` field by pressing TAB repeatedly. `pen` will blink (on most terminals) to indicate that the cursor is on it. Use the space bar to cycle the cursor through the `pen`, `area`, and `both` fields. Press NL to select a scope; the selected field will be displayed in reverse video (on most terminals). If the cursor is not in a display area, you will be able to set attributes only for the pen.

The pen is a new feature in release 5. Setting attributes for the pen means that, from that time on, all display text on this screen will be created with the specified attributes. By changing the pen attributes, you can create adjacent text with different display attributes. Note that creating contiguous text with different display attributes actually creates different display areas.

Note that "reverse video" items have only one color. A compatible background color is automatically chosen.

Fields

Fields are areas of the screen that can hold application data. The data within fields are the only screen data which the underlying **JAM** application can access. Fields may be established individually, or combined into groups or menus (see page 83).

The data in fields can be accessed and/or modified in a number of different ways, depending on how the fields are specified by the developer:

- The user can manipulate a field within the limits imposed by the developer via field protection (see page 41).
- Field data is automatically moved between fields and the local data block.
- Attached hook functions (see page 59) can manipulate field data.
- JAM developers can place initial data in fields when the application is created.

Each field must be contained within a single line of the screen. Field creation is a two–step process. First, type draw field symbols (usually underscores) where you want to

create the field. Then, to translate these field place–holders into real fields, hit the XMIT key to *compile* the screen.

When a screen is compiled, any areas containing underscores (or other draw field symbols) that are not within an existing field or side border are converted into fields. All the fields are then renumbered from left to right, and then from top to bottom, starting with 1. Screens are automatically compiled whenever any of the following happens:

- The PF2 key is pressed to toggle from draw to test mode.
- The XMIT key is pressed.
- The EXIT key is pressed, resulting in display of the Screen Editor Exit window.

A field may contain initial data that is displayed whenever the screen is opened. Initial data is entered in draw mode and, if the field is unprotected, in test mode by simply typing the initial data into the field. It is often useful to use the ZOOM feature when entering initial data into shifting fields and arrays in draw mode. Note that, when a screen is opened, the initial data will override LDB data for a field. See the Authoring Reference chapter, page 135, for more information about the interaction between the LDB and fields on screens.

For each field, a number of characteristics may be defined. Field characteristics affect screen behavior. Characteristics include:

- *Identifiers* *Field names* and *field numbers* identify fields to the Screen Manager, to the **JAM** Executive and to the application. However, the local data block recognizes only field names. Field numbers are assigned automatically when the screen is compiled and are not under the control of the author. Field names are assigned and maintained exclusively by the author. Reference by field name is generally safer than reference by field number, because field numbers can change each time the screen is changed.
- *Sizes* A field's size is measured in terms of its on–screen width, and its total on–screen and off–screen width. A *shifting field* is a field whose off–screen width is greater than zero. In addition, a field is part of an array. An array's size is measured in terms of its number of occurrences and elements. The number of elements is the number of fields (fields are always on–screen) in the array. The number of occurrences is the array's total capacity (on– and off–screen). The number of populated occurrences is the occurrence number of the highest array occurrence containing data. The default size characteristic is an array of one element and one occurrence for a simple field.
- *Attributes* Attributes include character and field level edits that restrict, format, and validate data input. Attributes also affect the appearance of fields and field data.

- *Attachments.* Attachments include JPL procedures and C functions that are invoked at field entry, exit, or validation. They also include help screens, status text, item selection screens, and next/previous field ordering specifications.

You can delete, move, and copy fields. When a field is moved, all of the field's characteristics are moved with it except its field number, which is dependent on the field's location relative to other fields on the screen. When a field is copied, the same is true except that the new field is unnamed, because field names must be unique within a screen.

4.3.2

Screen Characteristics (PF3)

From either test or draw mode, pressing PF3 displays the Screen Characteristics Screen shown below.

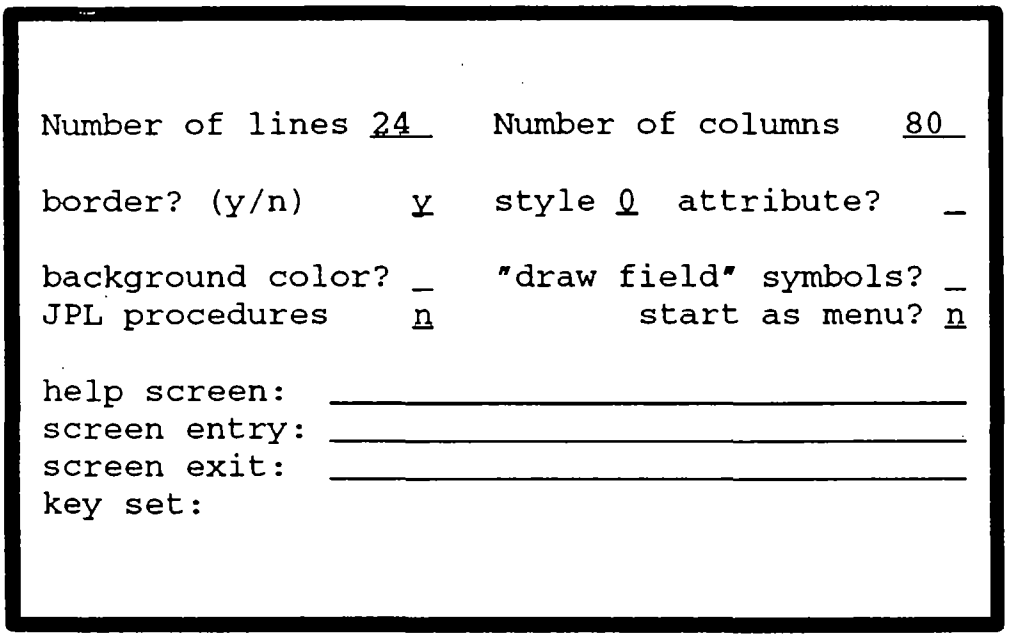


Figure 13: Screen Characteristics Screen

Screen characteristics are changed by modifying the appropriate fields. In some cases (e.g. `attribute`), an additional window is displayed with additional characteristics. To view the additional window, enter a `y` in the field. Press XMIT to confirm changes made to a particular window and to close the window. Press EXIT to cancel changes made to that window and to close the window. The details of the screen characteristics window are explained below.

Screen Size

By default, **JAM** sets the size of any screen you create to be the size of the physical display, less one line for status (unless the terminal has a separate status line) and one or two lines if simulated soft keys (see the Keyset Editor chapter, page 119) are used. Since many standard displays have 24 lines, it is often good to limit the screen size to 23 lines (21 or 22 if simulated soft keys are used). On the other hand, since **JAM** screens are virtual screens, you can create screens as large as 254 lines by 254 columns. When a virtual screen is larger than the physical display, then the screen is viewed through a mechanism called a *viewport* (page 152). You can also create screens that are smaller than the display for use as pop–up windows.

To change the size of the current screen, modify the fields labelled `Number of lines` and `Number of columns`. The specified dimensions are the dimensions of the virtual screen, not of the viewport. Viewport size and position are not pre–determined screen characteristics. They are specified when screen display is requested. See the chapter on control strings in the Authoring Reference Chapter for information about setting viewport size and position.

Screen Border

A border around a screen can increase its visibility, especially if it is used as a pop–up window. By default, **JAM** creates screens without borders. A border has a style (numbered 0–9) and display attributes (such as intensity, foreground color, and background color)[6].

Type `n` in the `border (y/n)?` field to delete an existing border. To create or modify a border, type `y` in this field. Additional fields will appear: `style` and `attribute`.

The default border style is 0. To change the border style, type the number of the desired style into the `style` field. As you change the border style, the style of the border of the Screen Characteristics Screen itself changes, allowing you to sample the styles.

The default border attributes are reverse video highlight with a color of white. To change display attributes, type `y` in the `attribute?` field. The Display Attributes Screen (see page 24) will pop up as shown below so that you can change the border's display attributes.

6. The border style is an index, and not a particular character set, stored with the screen. Therefore, by editing the video file and running the `vid2bin` utility described in the *Utilities Guide*, the actual character set associated with a particular border style index can be changed, and the look of the screen can be changed without ever entering the Screen Editor.

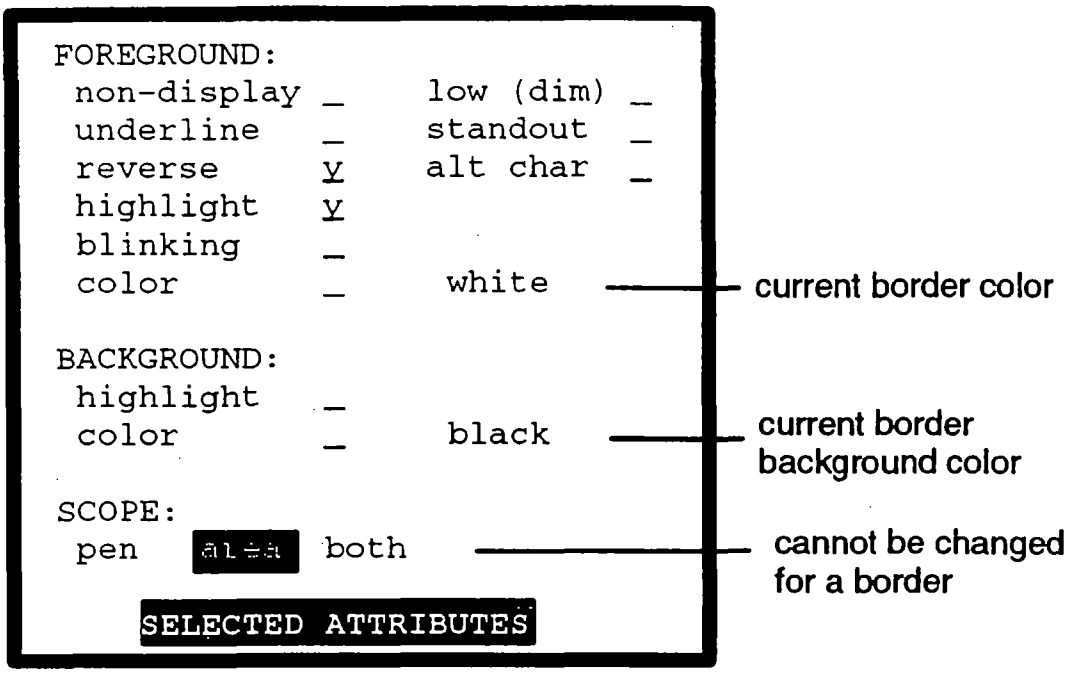


Figure 14: Display Attributes Screen For Border

Screen Background Color

The default background color is black with no highlighting. To change the screen's background color, type y into the `background color?` field. The Display Attributes Background Color Screen will pop up as shown below.

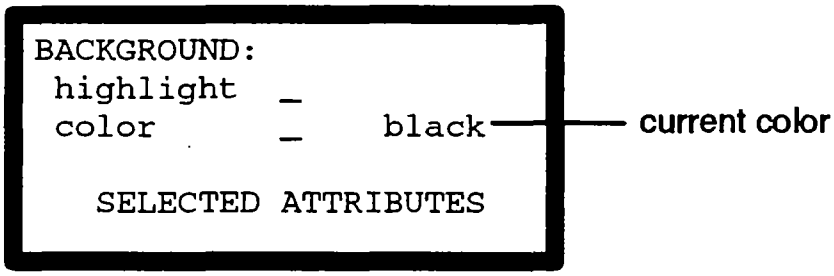


Figure 15: Display Attributes Background Color Screen

To specify a highlighted color, enter y into the `highlight` field. To specify the color itself, enter y into the `color` field. The Color Menu will pop–up (see Figure 12 on page 25).

Field Drawing Symbols

The default field drawing symbol is the underscore. A field is painted onto the screen by typing underscores and pressing XMIT. By default, the field is unprotected, underlined, and highlighted. The developer then assigns edits and attachments to the field. As you will see in following sections, the process of fully characterizing a field can be a rather long one. Therefore, you can create up to nine field drawing symbols, each with its own set of default characteristics. To use the same draw field symbols on several different screens, define them on one screen and use that screen as a template for the others.

To specify draw field symbols, type `y` in the `draw field symbols?` field. The Draw Field Symbols Screen, pictured below, will pop up. Initially, only the underscore will be defined; you can define up to eight more symbols in the spaces provided. To add or change a draw field symbol, position the cursor inside a pair of brackets and type the desired symbol.

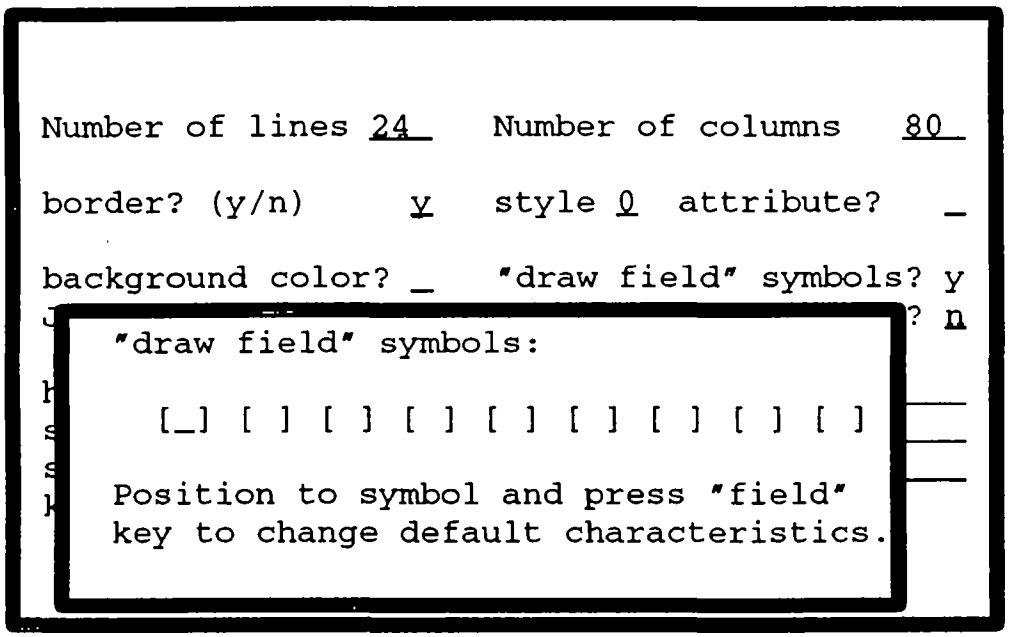


Figure 16: Draw Field Symbols Screen

To assign field characteristics to a symbol, move the cursor to that symbol and press the PF4 key. The Field Characteristics Screen will pop up. You can assign field characteristics to the symbol, as described later in this chapter.

Screen–level JPL Procedures

Procedures written in JPL, JYACC Procedural Language, may be stored in the JPL Procedures Screen that is attached to a screen. These procedures can be called whenever the screen is active (i.e. is the top–level screen). The advantages of storing JPL procedures in a screen's JPL Procedures Screen are:

- The JPL is partially syntax–checked and compiled when the JPL Procedures Screen is closed.
- The JPL is stored with the screen, rather than in a separate file.
- The first procedure, if nameless, is called when the screen is opened. This enables initialization of variables, including JPL variables, screen fields, and Local Data Block entries.

The *JPL Guide* contains more complete information about JPL procedures. Note that you can test screen–level JPL procedures in application mode, but not in test mode.

To enter or modify JPL procedures, type `y` into the `JPL Procedures?` field. The JPL Procedures Screen will pop up as shown below, enabling JPL entry and editing.

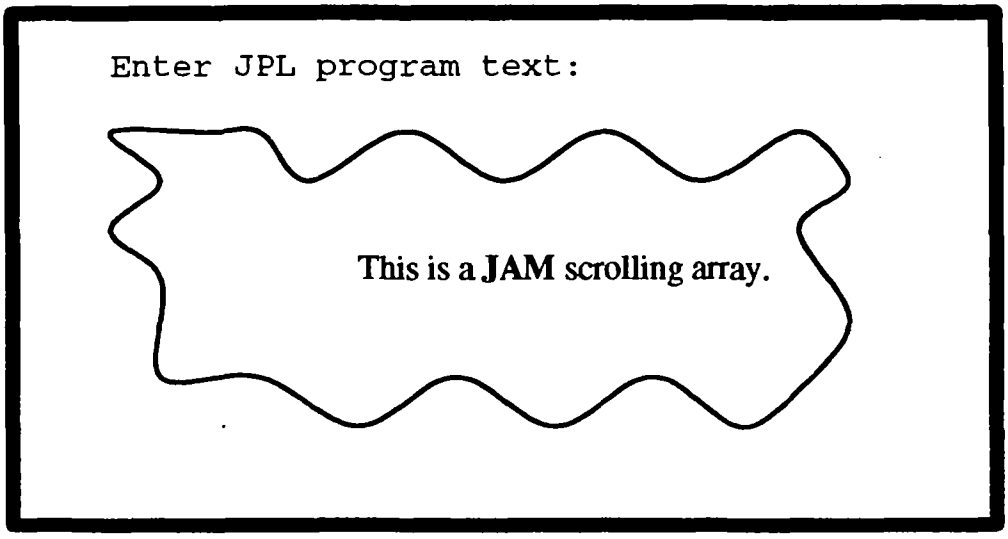


Figure 17: JPL Procedures Screen

Since the screen contains a **JAM** scrolling array, you may use the **JAM** keys DELETE LINE and INSERT LINE to delete and insert lines. As in other Screen Editor screens, you must press XMIT to accept the changes made to the JPL. Pressing EXIT will discard all changes.

It is often convenient to use a text editor to create JPL, and then to use the PF4 feature to update the JPL Procedures Screen. You can read and write JPL files from within the JPL Procedures Screen by pressing PF4 to display the JPL File Operations Screen shown below.

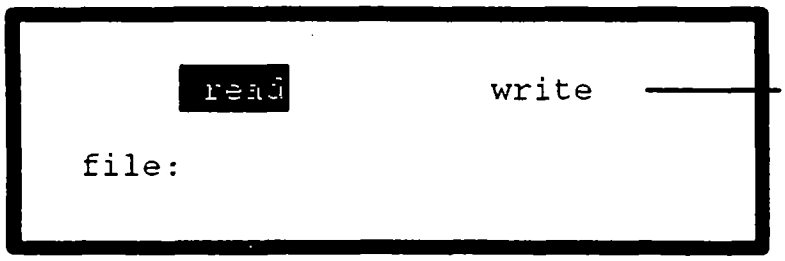


Figure 18: JPL File Operations Screen. The screen has a read/write radio button and a data entry field for specifying the file name.

Select `read` to import text at the current cursor position, or `write` to export text, and type in the name of the file in the field provided. Press XMIT to perform the selected operation, or press EXIT to abort the operation.

Starting Mode for Runtime Screen

A screen can behave both as a menu and as a data entry screen, but not at the same time; menus are active only when the screen is in *menu mode*, while data entry is possible only in *data entry mode*. When a screen has menu fields and no other unprotected fields, it is always in menu mode. When a screen has no menu fields, it is always in data entry mode. However, if a screen has both menu fields and unprotected data entry fields,the end user can toggle the mode by using the MENU TOGGLE key or a mouse click. In that case, JAM's default is to start the screen in data entry mode.

To start a screen in menu mode, type `y` in the `start in menu mode?` field. The field is ignored at runtime, unless the screen has both menu fields and unprotected data entry fields.

Screen Level Help

You can designate a help screen for the screen as a whole. Help screens are always displayed as windows so as not to destroy the underlying screen. The help screen is displayed when the SCREEN HELP key is pressed or when the HELP key is pressed and the cursor is not in a field with its own help or item selection screen.

To designate a help screen for the screen you are currently editing, move the cursor to the `help screen` field and enter the name of the help screen. You can optionally

precede the screen name with the row and column, in parentheses, at which the help screen is to appear. In the following example, the screen `hints` will be displayed at row 5, column 25:

```
help screen: (5,25)hints
```

Additional positioning parameters may be specified; please see the discussion on positioning field help screens (page 53). See the Authoring Reference for information on the design of help screens.

Screen Entry and Exit Functions

Screen entry and exit functions, also referred to as screen functions, are used to process data at the time of screen entry and exit. Screen functions can be written in C or JPL. The *Programmer's Guide* describes the default arguments passed to screen functions.

To specify a screen function written in C, enter the function's name and arguments in either the `screen entry` field or the `screen exit` field. To specify a screen function written in JPL, enter the word `jpl`, followed by the JPL procedure name and arguments. *Colon preprocessing* (see page 121) is performed on the function's arguments: any argument preceded with a colon is assumed to be a field name (or a local data block entry name, or a group name) and is replaced with the contents of the field. The following example shows a screen entry function named `setup` written in C, and a screen exit function named `cleanup` written in JPL.

```
screen entry: setup
screen exit: jpl cleanup
```

Caution: These are not control strings. Therefore, do not precede the function name with a caret(^).

A screen's entry function is called whenever the screen is made active. This occurs when the screen is opened or when a screen is made active with the VIEWPORT key. A screen's exit function is called whenever the screen is made inactive. This occurs when the screen is closed or when the screen is made inactive with the VIEWPORT key. **JAM** guarantees that the screen entry and exit functions will be paired; for every time the screen entry function is called, the screen exit function will be called exactly once.

For compatibility with earlier releases of JAM, the screen entry function is not called when a screen is made active by virtue of being exposed when a window overlying the screen is closed. Similarly, the screen exit function is not called when a screen is made inactive by virtue of being hidden when a window is opened that overlies the screen. You may alter this behavior so that screen entry and exit functions will be called when a window is exposed or hidden as follows. Either make the following library call (generally in `jmain.c`):

```
sm_option ( EXPHIDE_OPTION, ONEXPHIDE );
```

or include the following statement in your setup file:

```
EXPHIDE_OPTION = ON_EXPHIDE
```

There is a mechanism for calling the same screen entry and/or exit functions for all or most screens in an application. See the *Programmer's Guide* for details.

Screen Level Keyset

JAM has an optional set of keys known as *soft keys*. Soft keys allow the developer to create a set of special function keys with associated labels that appear on the screen monitor. The logical translation of a soft key varies, depending on the row of labels which appears on the monitor when the key is pressed.

A *keyset* supplies the translation of a soft key into a logical key. It also contains the text that is used to display the correct labels on the screen. Keysets are created using the *Keyset Editor* (chapter 6). You may have more than one keyset in an application, for example an application level keyset and a screen level keyset. The translation of soft keys will change depending on which keyset is currently in use.

To specify a screen level keyset, enter the name of the keyset in the `keyset` field. Whenever this screen is open, the specified keyset will be active, unless a window with its own keyset is opened on top of it, or the application program overrides it. When this screen is closed, the application level keyset, or another open window's keyset, becomes active. If neither of these is available, then the default keyset is activated (see page 117).

Soft keys and keysets are discussed at length in the Keyset Editor chapter, as well as in the *Programmer's Guide*.

4.3.3

Set/Change Field Characteristics (PF4)

Most field characteristics are set via the PF4 key. To modify a field's characteristics, place the cursor on the field and press PF4; the Field Characteristics Menu will be displayed, as shown below.

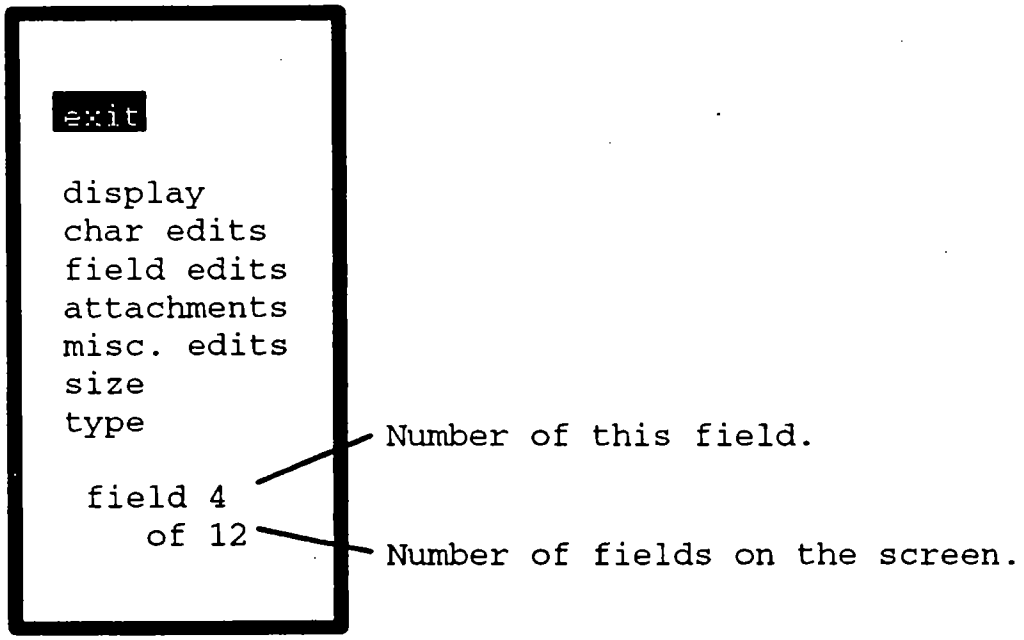


Figure 19: Field Characteristics Menu

The Field Characteristics Menu is the gateway to a hierarchy of menus and data entry screens[7]. The possible selections are briefly explained in the list below, and then detailed in the sections that follow.

- `display`
  Appearance of a field and its data. Note that the display characteristics of a field can be different than the display characteristics of an occurrence, in which case the occurrence's characteristics prevail (see `sm_achg` in the *Programmer's Guide*).
- `char edits`
  Restrictions on enterable characters, enforced as each character is typed.
- `field edits`
  Restrictions on enterable data, enforced when field is validated. Also, field protection and upper/lower case mapping.
- `attachments`
  Field name, tab–ordering, help screen, and item selection screen.
- `misc. edits`
  Attached functions, currency formatting, range checking, math calculations.

7. This menu hierarchy is the same one used when setting characteristics for draw field symbols and for data dictionary entries.

- `size`
  On–screen and off–screen field width. Number of array elements and occurrences.
- `type` C language data type to be used for a field when the field is copied into a C data structure.

To set field characteristics in any of the above categories, make the corresponding menu selection. Note that all of the fields of an array share the same set of field characteristics. To return to draw or test mode, press the EXIT key, or choose the exit option on the menu.

Field Display Attributes

By default, fields are underlined and highlighted on display terminals that support those attributes. JAM "fakes" underlining with underscores on terminals that do not support underlining. The default foreground and background colors are white and black. To change the display attributes of a field, choose `display` from the menu. The Display Attributes Screen will appear as shown below. The Display Attributes Screen is described fully on page 24.

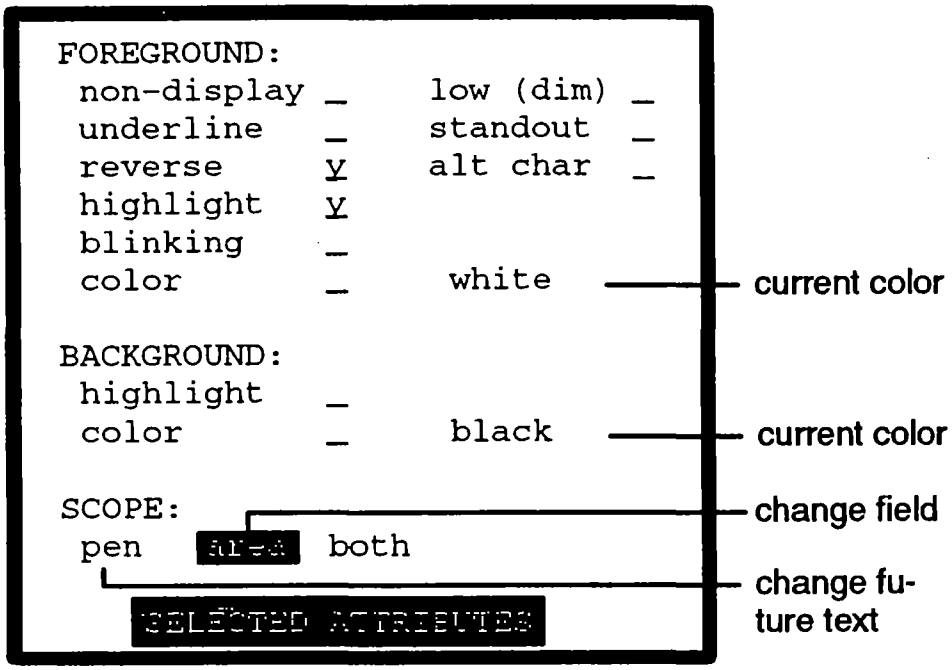


Figure 20: Display Attributes Screen For Field

Character Edits

Character edits provide a mechanism to filter field input on a character–by–character basis. If the end user attempts to enter a character into the field which does not match

the character edits specified, the warning bell will ring and that character will be rejected. By default, the character edits for a field are set to unfiltered, meaning that any character is accepted when typed.

To place a character edit on a field, choose `char edits` from the menu. The Character Edits Menu will appear as shown below. Exactly one character edit must be in force at all times; that edit will be shown in reverse video.

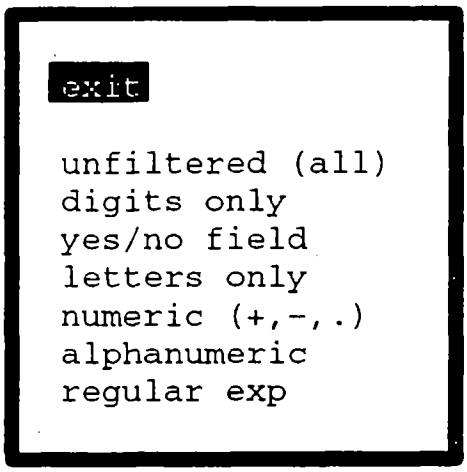


Figure 21: Character Edits Menu

To change the edit, select a new edit from the menu by typing the first letter of its name or by positioning the reverse video bounce bar and pressing XMIT.

The available character edits are described below:

- `unfiltered`
  Allows entry of all characters with no restrictions.

- `digits only`
  Allows entry of the digits 0–9 only. If left–justified, no blanks are allowed to the left of any digit. If right–justified, no blanks are allowed to the right of any digit. Therefore, when a digit is entered into the middle of an otherwise empty field, it is rejected (due to the leading/trailing blanks). This edit permits the use of embedded punctuations as explained below.

- `yes/no field`
  Allows entry of only the initial letters of "yes" and "no". This edit uses the initial letters, in upper and lower case, of the SM_YES and SM_NO entries in the message files. By default, this allows `y`, `Y`, `n`, `N`, or space, which is converted to n. The message file can be changed to support non–English equivalents to `Y` and `N`.

- `letters only`
  Allows only a–z, A–Z, and the space character. JAM actually uses the C

library function `isalpha` which, on some systems, includes characters such as Ü. This edit permits the use of embedded punctuations as explained below.

- `numeric`

  Allows entry of digits, the plus or minus sign, and at most one decimal point. If present, the plus or minus sign must be leftmost in the field. If left justified, no blanks are allowed to the left of any character entered. If right justified, no blanks allowed are to the right. Therefore, when a valid character is entered into the middle of an otherwise empty field, it is rejected (due to the leading/trailing blanks). This edit permits the use of embedded punctuations as explained below.

- `alphanumeric`

  Allows entry of any digits, the letters a–z and A–Z, and the space character. **JAM** actually uses the C library functions isalpha and isdigit.

- `regular exp`

  Allows entry of characters that match a *regular expression* (a pattern), entered by the author in the Regular Expression window that pops up when this selection is made. For example, the regular expression `[A-Z].*` matches any string that begins with a capital letter. There is a detailed description of regular expressions in the Authoring Reference chapter.

Embedded Punctuation

A special feature of digits–only, letters–only, and alphanumeric fields is that punctuation characters within the field are passed over during normal data entry[8]. The punctuation characters themselves must be entered in draw mode; they can not be entered or deleted in test mode or at application run time[9]. This feature is useful for telephone numbers, social security numbers, and other punctuated strings. For example, consider a field for the entry of a telephone number of the form ***nnn/nnn–nnnn***. We don't want the user to enter the punctuation characters / and –. Create the field in draw mode by typing 12 underscores. Press XMIT to compile the underscores into a field. Press PF4, followed by `c` and then `d` to make the field be digits only. Press XMIT to return to draw mode. Enter / and – in the fourth and eighth positions respectively. The field should look like:

```
___/___-____
```

8. **JAM** uses the C library function `ispunc` to determine whether or not a character is a punctuation character. On some systems, `ispunc` considers the space to be a punctuation character, but **JAM** never considers the space to be a punctuation character.

9. The **JAM** library function `sm_putfield` can be used to change (or delete) the punctuation characters at runtime if its data argument is anything other than a zero length character string.

Press PF2 to enter test mode. The punctuation characters will remain in place while you type digits — even if you press the FIELD ERASE key.

Embedded punctuation also works within scrolling arrays. When a new array occurrence is allocated, embedded punctuation is copied from the first occurrence of the array to the newly allocated occurrence. Therefore, as the array scrolls, every enterable field is correctly initialized with the embedded punctuation. See the Scrolling Array section (page 144) of the Authoring Reference for a discussion of scrolling arrays and allocation of occurrences.

Field Edits

Field edits validate and format data keyed into a field. Field edit validation differs from character edit validation in that entry is rejected not on a character by character basis, but according to the contents of the field as a whole. With the exception of upper and lower case translation, field edits are not mutually exclusive. Therefore, they are specified by a data entry screen with yes/no fields. By default, **JAM** creates fields with no field edits in effect.

To modify the field edits for a field, choose `field edits` from the menu. The Field Edits Screen will appear as shown below. The Field Edits Screen is a list of edits that can individually be turned on or off by typing `y` or `n` after them. When the Field Edits Screen is displayed, all field edits that are currently in force are followed by a `y`.

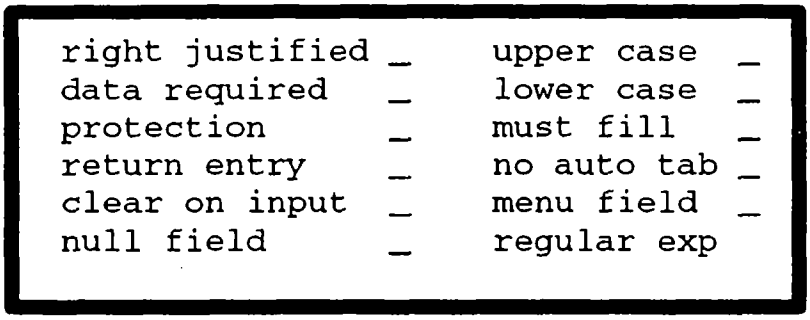


Figure 22: Field Edit Screen

To enable an edit, type `y` after it. To disable an edit, type `n`. Certain edits, such as protection, have a submenu for specifying more detailed information. The submenu is displayed only when `y` is typed, even if the `y` is already present; when you tab through the field, you must re–type `y` to get the submenu. When you are satisfied with your selection of field edits, press XMIT to effect the change. If you press EXIT instead, the field edits will stay as they were, except that changes made on submenus remain in effect. The field edits are discussed in detail in the sections that follow.

Right Justified Fields

By default, **JAM** fields are left justified, and characters are entered from left to right one after the other across the field. If you specify that a field is to be right justified,

characters are entered starting at the rightmost position of the field; the rightmost position is considered to be the beginning of the field. As each additional character is entered, the previously entered data is shifted one position to the left. When you tab into a right–justified field, the cursor is positioned at the rightmost position.

There are some idiosyncrasies of right–justified fields that should be noted:

- The FIELD ERASE key clears the entire field, instead of from the cursor to the end of the field.
- Insert mode is implicitly on in the rightmost position. Therefore, **JAM** will beep if you attempt to enter a character into the rightmost position of a completely filled field. Use the clear–on–input edit to force re–typing of the entire field.
- When restricted under character edits to digits–only or numeric–only fields, there can be no blanks to the right of any character.

Data Required Fields

You may want to force a user to enter certain information on data entry screens. When a field is specified to be data required, then it must contain at least one non–blank character before it can be tabbed from by the end user. If it is a digits–only, numeric, or alphanumeric field, it must have at least one character in it; punctuation characters entered in draw mode do not count. If the field is left blank by the end user, **JAM** displays an error message and repositions the cursor to the beginning of the field. Note that entry in array fields is required only for the allocated array occurrences, which always includes the array elements. See the Scrolling Arrays section (page 144) of the Authoring Reference chapter for a discussion of when occurrences are allocated.

If a data required field also has a null edit, then the field must be non–null in order to pass validation (the null indicator string does not satisfy the requirement for data).

Field Protection

A field can be protected from data entry, tabbing into, clearing, and validation. Typing `y` in the protection field will cause the Field Protection Screen to be displayed, as shown below. This screen allows you to specify any combination of field protections by entering a `y` or an `n` after each type of protection.

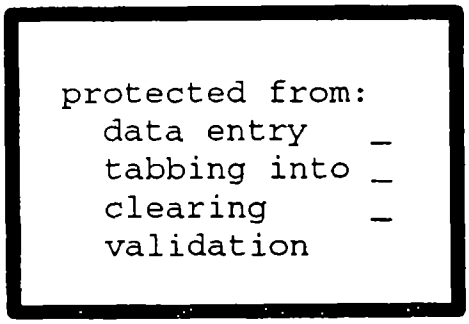


Figure 23: Field Protection Screen

Each type of protection is explained below:

- `data entry`
  All characters typed into the field will be rejected with a beep. The CLEAR ALL, FIELD ERASE, DELETE CHAR, DELETE LINE, and INSERT LINE keys will not work in the field. DELETE LINE and INSERT LINE will work in a field parallel to one protected from data entry, as long as it is not also protected from clearing. In Figure 24, the ***extprice*** field is protected from data entry, but unprotected from clearing, so INSERT LINE and DELETE LINE will work in the parallel fields. The **JAM** library functions may still enter data into fields protected from data entry.

- `tabbing into`
  The cursor cannot be moved into the field by the user. **JAM** library functions can still move the cursor into the field.

- `clearing`
  The CLEAR ALL, FIELD ERASE, DELETE LINE, and INSERT LINE keys will not clear the contents of the field. Certain **JAM** library functions can be used to clear the field.

- `validation`
  Field validation will not be performed, even when the screen as a whole is validated. Character edits will still be enforced. **JAM** library functions cannot be used to validate a field that is protected from validation.

A noteworthy combination is protection from data entry , but not from tabbing. This combination is recommended when a scrolling or shifting field should not be modified, but must be shifted or scrolled in order to view the field's entire content.

A field derived from other fields on the screen (e.g. extended price is derived from quantity and unit price) can be protected from everything except clearing. This permits the user to clear the derived field, while prohibiting direct (and possibly erroneous) changes to the field. For example, consider the synchronized arrays (i.e., they scroll together: see page 70) protected on a screen as shown below:

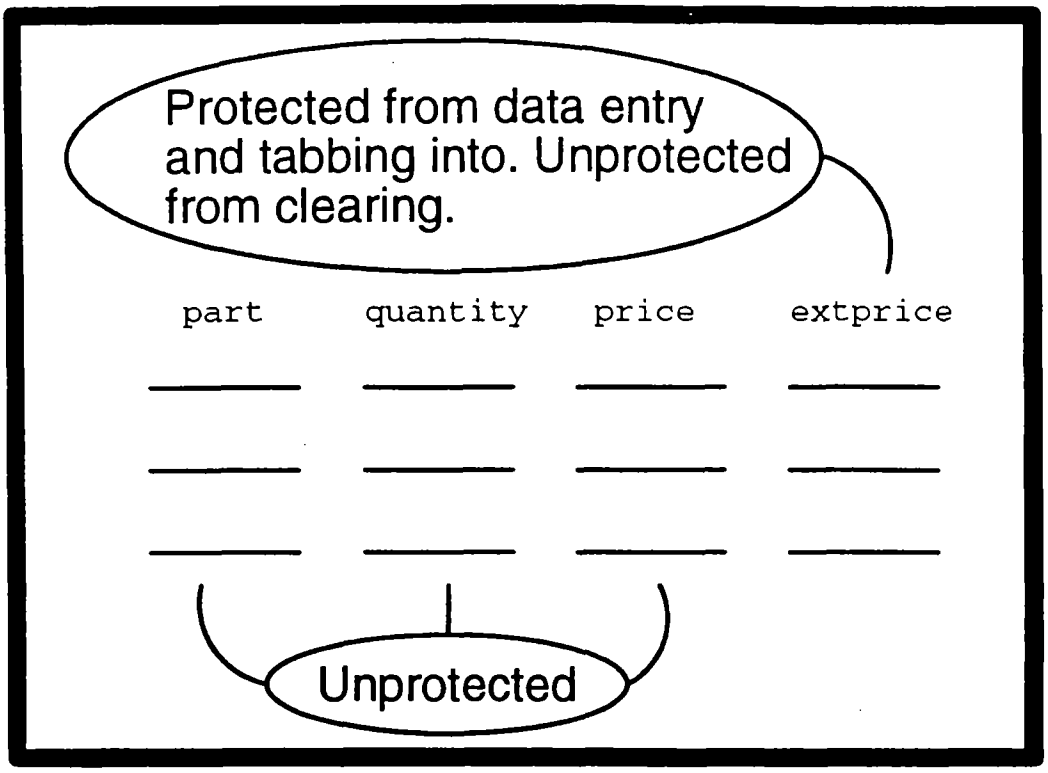


Figure 24: Protection Example

On the above screen, the user cannot directly change `extprice`.The user can clear an entire row by pressing the DELETE LINE key. This works because the arrays are synchronized and because `extprice` is not protected from clearing. CLEAR ALL will work also in this case.

For groups, protecting a field within the group from data entry and clearing means that the user cannot select an un–selected field or de–select a selected field.

Return Entry Fields

Normally, on data entry screens, the Screen Manager returns control to the **JAM** Executive when a function key is pressed. The **JAM** Executive processes the associated control string, if one exists. If you designate a field to be return entry, the Screen Manager will return control to the **JAM** Executive whenever the field is filled or tabbed out of.

Typing `y` in the return entry field will pop up the Return Code Screen as shown below. This screen has one field on it, which is an integer that the Screen Manager will return to the **JAM** Executive.

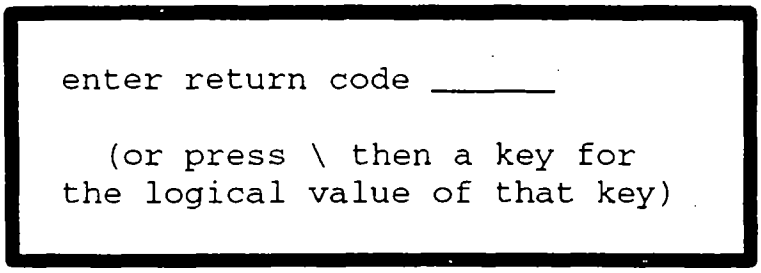


Figure 25: Return Code Screen

The return code may be entered in any of several formats:

- A decimal integer, like `50`.
- An octal integer with a leading zero, like `062`.
- An hexadecimal integer with a leading `0x`, like `0x32`.
- An ASCII character, with surrounding apostrophes, like `'S'`.
- A **JAM** key mnemonic, as an alphanumeric string. As indicated on the window, you may also press the \ key, followed by the desired function key, to generate the mnemonic automatically.

If you do not specify a return code, the Screen Manager will return the ASCII value of the rightmost character in the field. In general, unless you are writing your own executive, return entry fields will be useful only if the return code is a **JAM** logical function key, such as XMIT, EXIT, PF1, PF2, etc. In such cases, the application will act as if the user pressed the logical function key, except that XMIT will not cause validation of the fields on the screen.

Clear on Input

In a field designated clear on input, the field is automatically cleared of previous data whenever a new character is typed in the beginning position. This feature is useful for right–justified or currency fields, in which overwriting previous data might be confusing for the user. It is also useful for fields that tend to change completely, if they change at all. Note that a clear on input field is not cleared if the user moves beyond the beginning position and types a character.

Null Field

The null field edit enables a field to be null by providing a special string, called a null indicator string, that differentiates a null field from a field that is not null (e.g. from a blank field). When you designate a field to have a null field edit, the Null Indicator Screen is displayed as shown below.

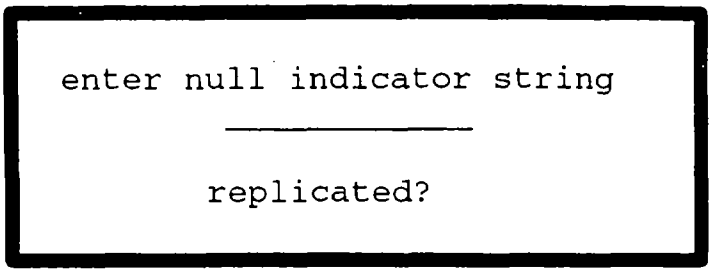


Figure 26: Null Indicator Screen

There are two enterable fields on the Null Indicator Screen. The first field, labelled `enter null indicator string`, is where you enter the null field string. Up to 256 characters may be entered. The second field, labelled `replicated`, is a yes/no field. If you enter `y` in that field, the null indicator will be repeated to fill the field. For example, specifying a replicated null indicator string of a single asterisk (*) would fill a null field with asterisks, up to the length of the field. If the null indicator string is not specified, then both the null indicator string and replication are taken from the message file. This provides a mechanism for specifying an application–wide (and language customizable) default.

The following conditions must hold for a field to be considered to be null:

- The field must have the null field edit.
- The field must contain the null indicator string.

A user can make a field (with the null edit) null by clearing the field with FIELD ERASE or by entering the null indicator string (replicated, if replication is specified) into the field. Using the space bar to blank the field will not make the field null, it will simply make the field blank. In addition, a field that contains no initial data and is not populated by data from the LDB (except for the null indicator string) will be displayed as a null field when the screen is displayed.

The null indicator string will be cleared under exactly the same circumstances that a clear–on–input field would be cleared — when data is entered into the first position of the field before being entered elsewhere in the field.

Programmers should use the function `sm_null` to determine whether or not a field is null. The function `sm_getfield` will return the content of a field, which is the null indicator string (possibly replicated) in the case of a null field.

Upper and Lower Case Fields

These edits translate data entry to upper or lower case as each character is typed. Non–alphabetic characters are not affected. Upper and lower case translation are mutually exclusive. Applying one of these edits to a screen name field (see page 88) causes the screen name to be displayed in the specified case. See the *Programmer's Guide* for internationalization considerations.

Must Fill Fields

A field designated must fill is considered valid in the following cases:

- if the field is completely empty, or
- if the field has no blanks. This includes leading, trailing, and embedded blanks.

If a field must be filled (i.e. an empty field is not acceptable), then it should be assigned both the must fill and data required edits.

No Auto Tab Fields

Normally, when a user fills the last position of a field, the cursor will behave as though the TAB key were pressed. The field is validated and, if it passes, the cursor jumps to the beginning of the next field. For fields specified as no auto tab, however, the user must use TAB, NL, or some other cursor motion key to leave the field. The cursor will remain in the last position, and as each character is typed, it will replace the previous character in the last position. This edit is ignored for fields in groups, as they have their own internal auto tab edit.

Menu Fields and Submenus

A field must have the menu field edit in order to be menu selectable. Such a field is called a menu selection field. The menu field edit has no impact on a screen's behavior when the screen is in data entry mode. For a complete discussion of menu building, see the menu section of the Authoring Reference, chapter NO TAG A shortcut method of building a menu is described later in this chapter (see SPF2 on page 84); the shortcut method automatically creates the menu selection fields. The purpose of this section is to discuss the effect of the menu field edit, and to discuss submenus.

Typing `y` in the menu field will cause the Menu Field Screen to be displayed, as shown below.

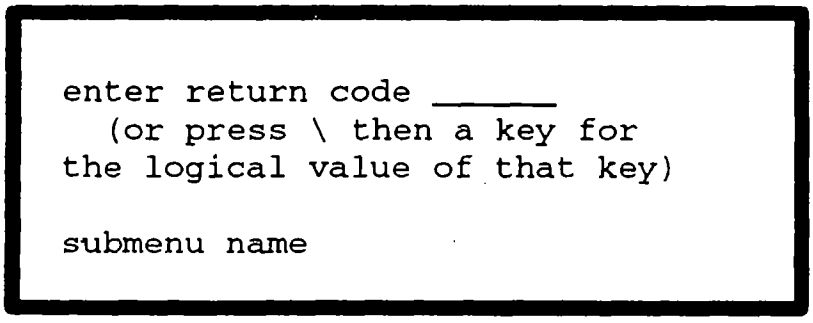


Figure 27: Menu Field Screen

The Menu Field Screen allows you to specify a return code or a submenu. In order for the menu selection to cause an action (when using the **JAM** Executive), at least one of the following items must be present:

- A control string in the field following the menu field. The control string is ignored if a submenu is specified; the action to be taken will be specified on the submenu.
- A return code. The return code is ignored if a submenu is specified; the return value will come from the submenu.
- A submenu.

The return code may be entered in any of several formats:

- A decimal integer, like `50`.
- An octal integer with a leading zero, like `062`.
- An hexadecimal integer with a leading 0x, like `0x32`.
- An ASCII character, with surrounding apostrophes, like `'S'`.
- A **JAM** key mnemonic, as an alphanumeric string. As indicated on the window, you may also press the \ key, followed by the desired function key, to generate the mnemonic automatically.

If you do not specify a return code in the window, the Screen Manager will return the ASCII value of the leftmost character in the field. In general, unless you are writing your own executive, the return code will be useful only if it is a **JAM** logical function key, such as XMIT, EXIT, PF1, PF2, etc. In such a case, the application will act as if the user pressed the logical function key, except that XMIT will not cause validation of the fields on the screen.

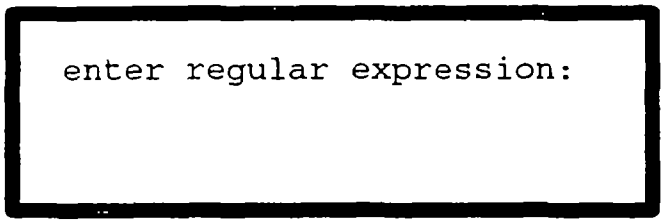
In the submenu field, you can type the name of another screen to serve as a pull down menu for the selection. Submenus work as follows: When the cursor enters a menu field with a submenu, the submenu window is automatically pulled down. All cursor motion keys operate normally in the submenu, except the left and right arrow keys; they move to the previous or next selection in the *main* menu, and pull down a new submenu if that selection has one. To create a main menu with submenus, create the main menu's selection fields from separate fields, not from fields in an array — otherwise each main menu field will share the same submenu. Since the shortcut menu building feature creates an array, it should not be used to create a menu that will have submenus. However, the shortcut method can be used to create the submenus themselves.

Regular Expressions

You can attach a regular expression (a pattern) to a field. For example, the regular expression `[^A-Z][A-Z]*` matches any string that doesn't start with a capital letter, but

otherwise contains only capital letters. The contents of the field are compared to the regular expression when the field is validated, not as each character is entered as is the case with a regular expression character edit. Note that a field may have both a regular expression character edit and a regular expression field edit. **JAM** does not compare these edits for consistency, but you can use both regular expressions to your benefit. For example, you could use the character edit to restrict the field to a set of characters (e.g. `[0-9]`) and use the field edit to establish a pattern (e.g. `9..` to match any number between 900 and 999).

To enter a regular expression, type a `y` in the regular expression field. The Regular Expression Screen will display as shown below.



Use ZOOM to enter a long pattern.

Figure 28: Regular Expression Screen

Regular Expressions are discussed in detail in the Authoring Reference Chapter. Use the HELP key to review the syntax of regular expressions while in the Screen Editor. The syntax of the regular expression is checked when you press XMIT to save the expression and exit the Regular Expression Screen.

When deciding between using a regular expression in a character edit and a field edit, consider the complexity of the pattern. If the pattern is simple, then the character edit might be better because **JAM** immediately beeps when an invalid character is entered. However, if it is complex, then the user may not know why **JAM** beeped. If a field edit is used, then **JAM** displays a message, and positions the cursor to the invalid character when an error is encountered.

Field Attachments

Field attachments are specified in the Field Attachments Screen shown below.

The available field attachments are described in the sections that follow. Briefly, the attachments are:

- `field name`
  Name that identifies a field on a screen.
- `next field`
  Name or number of field to go to when TAB is pressed.

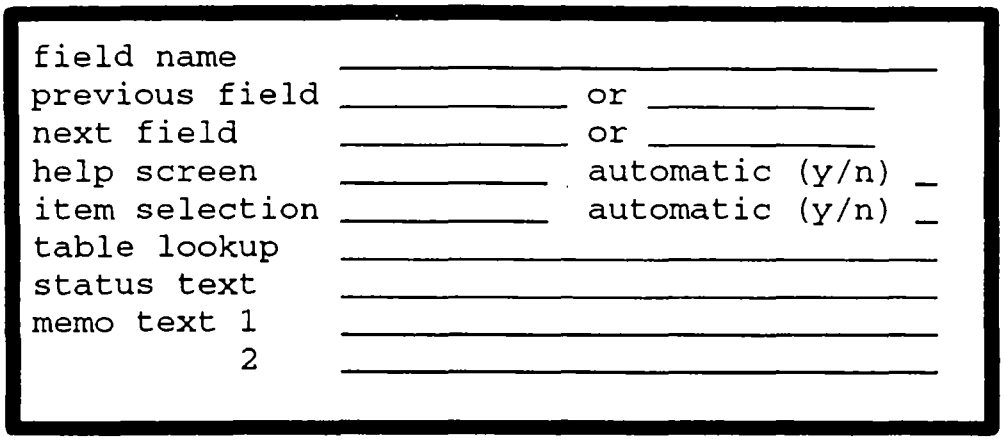


Figure 29: Field Attachments Screen

- `previous field`
  Name or number of field to go to when BACKTAB is pressed.
- `help screen`
  Window to display when field help is requested.
- `item selection`
  Window of items to choose from when field help is requested. Help screen and item selection are mutually exclusive.
- `table lookup`
  Screen containing list of acceptable field values used to validate field. An item selection screen may be used for table lookup.
- `status text`
  Help text displayed on the status line when the cursor is in the field. Status text can be used to display keystroke instructions (e.g. `Hit Enter`) in a keyboard independent fashion (e.g. `Hit %KXMIT`).
- `memo text`
  Up to nine lines of text for comments or for programmatic use.

Field Name

A field name identifies a field on the screen. No other field or group on the screen may share the name, except for fields in the same array; individual array occurrences are referenced by field name and occurrence number. A nameless field can be identified only by field number, which is tricky because the number can change automatically as the screen is edited. A field name can be used to refer to a field both within the Screen Editor (e.g. to specify the next field) and in application code. However, a field must be named if its contents are to be shared with the local data block.

When a field is copied, the copy retains all characteristics of the original except the name — otherwise a duplicate name would result. Fields on different screens can share

the same name, but it is good practice to share names only when the fields share the same data or field characteristics. In that way, data and characteristics can be shared through the local data block and the data dictionary respectively.

Field names can be up to 31 characters long, and must start with an alphabetic character, an underscore, a dollar sign, or a period. The rest of the name can contain alphanumeric characters, underscores, dollar signs, and periods. Field names are case sensitive, so that a field named `item` is different than a field named `Item`.

Previous and Next Fields

The default TAB ordering of fields (i.e. the order in which fields are visited by the cursor when TAB is pressed repeatedly, or when auto–tabbing occurs) is the same order in which the fields are numbered: left to right, and then top to bottom. The default BACKTAB ordering of fields (i.e. the order in which fields are visited by the cursor when BACKTAB is pressed repeatedly) is the reverse of the default TAB ordering. The next field and previous field entries enable changing the TAB and BACKTAB ordering of the fields on a screen.

The previous/next field designations have no effect on the cursor positioning keys or on the NL key.

A previous/next field designation is ignored if it refers to a field that is nonexistent or protected from tabbing into. In these cases, the previous/next field designation is said to fail. The Field Attachments Screen provides for two previous field designations and two next field designations; in either case if the first fails, the second is tried. If the second fails, the default field ordering is used.

Previous and next fields can be designated by field name or by field number. They may also be designated by group name, as described at the end of this section. Field numbers must be preceded by a # sign. For example, `#12` refers to field number 12. You can use relative referencing to refer to a field relative to the field number of the current field. Relative references must be preceded by a plus or a minus sign. The field just before the current field is `-1`. The fifth field following the current field is `+5`. To refer to the current field, use `+0` or `-0`.

You can designate a particular array occurrencesubparagraph by appending the occurrence number in square brackets to the field designation. Again, the number may be absolute or relative. The following are all valid previous/next field designations:

```
item
#23
+24 24th fld after current fld
-3
item[4]
item[-3]
+3[-4] 4th occurrence prior to the
       current occurrence of the 3rd
       field after the current field.
```

The following is not a valid designation:

```
item+1
```

Note that every occurrence of an array has the same previous and next field designation. If a designated previous/next field is also part of an array, *and* the designation contains no occurrence subscript, then the cursor is positioned in the designated array at the same occurrence number that it was in when it left the previous array. If the occurrence number is greater than the maximum number of occurrences in the designated field, the previous/next field operation fails, and the alternate, if one exists, is attempted.

For example, consider two horizontal arrays named `horiz1` and `horiz2` on the screen pictured below.

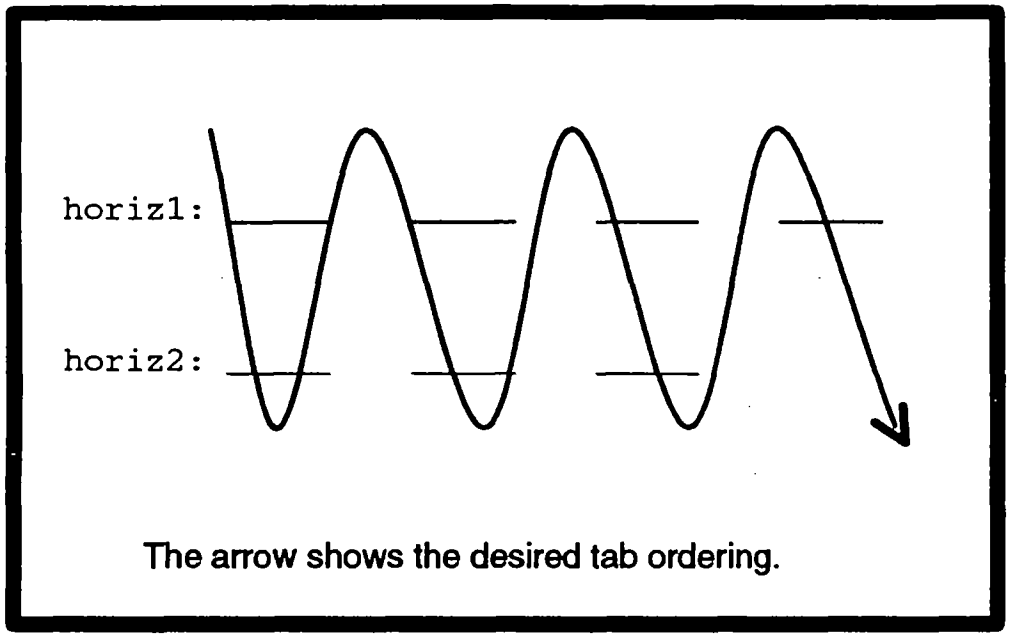


Figure 30: Next Field Example With Horizontal Arrays

By default the cursor would tab through `horiz1` before moving to `horiz2`. Let's modify the tab order so that tabbing moves the cursor from top to bottom, then from left

to right. To effect this, designate the next field for `horiz1` to be `horiz2` (with no subscript!), meaning "please tab next to horiz2, with the cursor in the same occurrence of `horiz2` as it was in `horiz1`". Designate the next field for `horiz2` to be `horiz1[+1]`, meaning "please tab next to `horiz1`, with the cursor in the next occurrence of `horiz1`".

As another example, consider two synchronized scrolling arrays (see page 70) named `array1` and `array2`, positioned on the screen as pictured below. Synchronized arrays scroll together.

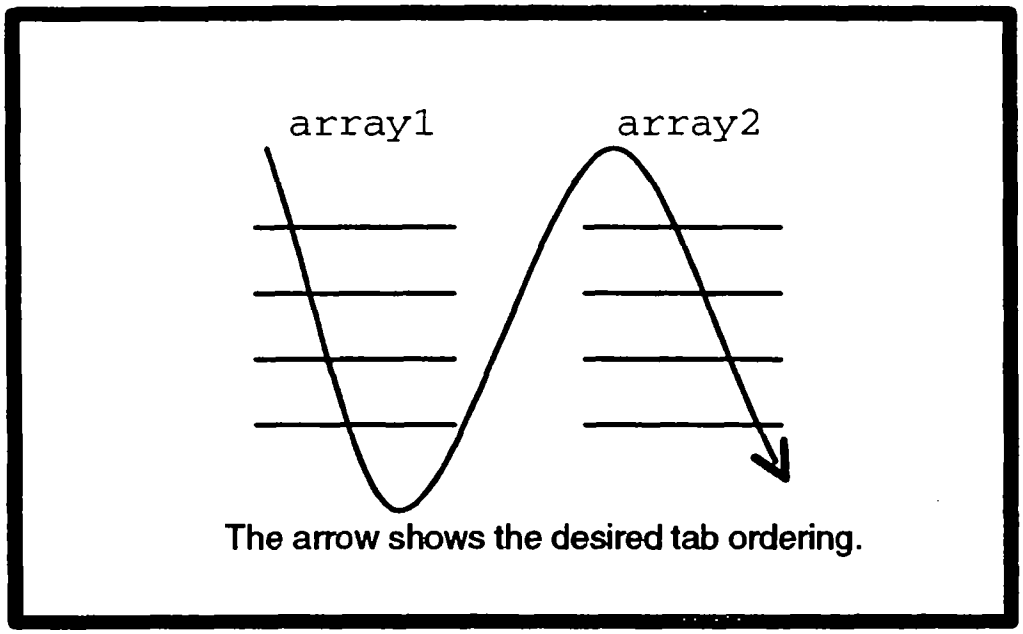


Figure 31: Next Field Example With Vertical Arrays

By default, the cursor would tab from the first occurrence of `array1` to the first occurrence of `array2` to the second occurrence of `array1` to the second occurrence of `array2` and so forth. Scrolling through the occurrences of an array would require pressing the NL or DOWN ARROW key. Let's modify the tab order so that tabbing moves the cursor through the screen in columns: all occurrences of `array1` and then all occurrences of `array2`.

To effect this, designate the next field for `array1` to be `array1[+1]`, meaning "please tab next to the next occurrence of this same array." Remember, every field of an array shares the same next field designation. In addition, **JAM** will scroll an array, if

necessary, to make offscreen occurrences visible[10]. When you reach the end of `array1` (just beyond the maximum occurrence number), that next field designation will fail. Therefore, designate the alternate next field for `array1` to be `array2[1]`, which will move the cursor to the top of the second array, and scroll both arrays so that their first occurrences are displayed in the top field of each array. Likewise, designate the next field for `array2` to be `array2[+1]`, and the alternate next field to be whatever field should be entered after tabbing through `array2`. Use the previous field edits to make BACKTAB have the opposite effect.

A group name (see page 85) may be specified as the previous and/or next field. This will cause the cursor to jump to the first field of the group. To cause the group to be entered in the Nth field of the group, append `[N]` to the group name. For example, `city[3]` refers to the third field in the city group.

Within a group, the next and previous field edits determine the next and previous fields when the SPACE and BACKSPACE keys are pressed (since these keys cycle through the fields in a group). The next and previous group edits (see page 92) determine where the TAB and BACKTAB keys take the cursor (since these keys cause the cursor to leave the group).

Help Screen

A field help screen can be displayed either when the user presses the HELP key, or automatically when the cursor enters a field that has not been marked as validated (see the Authoring Reference section on validation, page 150, for a discussion of when fields are marked as validated). To make the help screen display automatically upon field entry, type a `y` into the `automatic (y/n)` field. Note that a data entry field on the help screen automatically inherits the characteristics of the helped field, and its contents are transferred to the helped field. A detailed discussion of the JAM help facility, including help screen design techniques, is located in the Authoring Reference chapter (section 7.3).

To specify a field level help screen, enter the name of the help screen in the `help screen` field, optionally preceded by positioning parameters (also discussed in the reference in section 7.9) enclosed in parentheses. For example, the following entry specifies that the upper left corner of the `customer.hlp` screen is to be displayed at row 5 and column 10 of the physical display when help is requested:

```
help screen: (5,10)customer.hlp
```

The full format of the position specification is:

(row, col, height, width, vrow, vcol)

10. The TAB key, in this case, causes array occurrences to be allocated, if necessary, before scrolling the array. Normally, only the NL key has this effect. Please see the discussion in section 7.7 of the Authoring Reference (page 144).

where:

- ***row*** is the row of the physical display at which to position the upper left corner of the help window's viewport. If ***row*** starts with a + or – then ***row*** is the row offset, relative to the top left corner of the current screen, at which to position the upper left corner of the help window's viewport.
- ***col*** is the column of the physical display at which to position the upper left corner of the help window's viewport. If ***col*** starts with a + or – then ***col*** is the column offset, relative to the top left corner of the current screen, at which to position the upper left corner of the help window's viewport.
- ***height*** is size of the viewport in rows.
- ***width*** is the size of the viewport in columns.
- ***vrow*** is the row of the help window to be initially displayed in the upper left corner of the viewport.
- ***vcol*** is the column of the help window to be initially displayed in the upper left corner of the viewport.

If you do not specify a position, **JAM** will attempt to display the entire help screen without hiding the field. See section 7.9.2 on page 154 for more information on positioning. Note that there can be no ampersand (&) preceding the help screen entry, because the entry is not a control string.

Starting with **JAM** release 5, function keys work on help screens.

Item Selection

An item selection screen lets a user fill a field from a list of possible entries. The list can be static (created within the Screen Editor) or dynamic (created at runtime, possibly from a database query using **JAM/DB*i***, via a screen entry function or through the local data block). The list can either contain the complete set of valid entries, or a set of commonly used entries.

An item selection screen can be displayed either when the user presses the HELP key, or automatically when the cursor enters a field that has not been validated. To make the item selection screen display automatically upon field entry, type a `y` into the `automatic (y/n)` field. It is not possible to have both an item selection screen and a help screen for a field, although the Screen Editor does not detect the conflict.

When the item selection screen is displayed, the user can move to the desired entry and press XMIT to copy the entry back to the field. Pressing EXIT leaves the field unchanged. Item selection does *not* restrict data entry into a field, it only provides a list of

possible choices. To restrict the choices to the entries in the item selection screen use the table lookup feature described in the next section.

To specify an item selection screen, enter the name of the screen in the `item selection` field, optionally preceded by positioning parameters (also discussed in the reference) enclosed in parentheses. For example, the following entry specifies that the upper left corner of the `states.itm` screen is to be displayed at row 5 and column 10 of the physical display when help is requested:

```
item selection: (5,10)states.itm
```

If you do not specify a position, **JAM** will attempt to display the help screen without hiding the field. Note that there can be no ampersand (&) preceding the item selection screen entry, because the entry is not a control string.

An item selection screen may consist of any combination of fields, including scrolling and non–scrolling arrays. All should have the menu field edit. The lengths of the fields on the item selection screen need not be the same as the length of the associated field on the underlying screen. If the item selection fields are longer, as in the example below, only the first part of the field is copied. The rest of the field can contain information that is helpful to the user. If the underlying field has the right justified field edit, and is shorter than the item selection field, then the rightmost part of the item selection field will be copied.

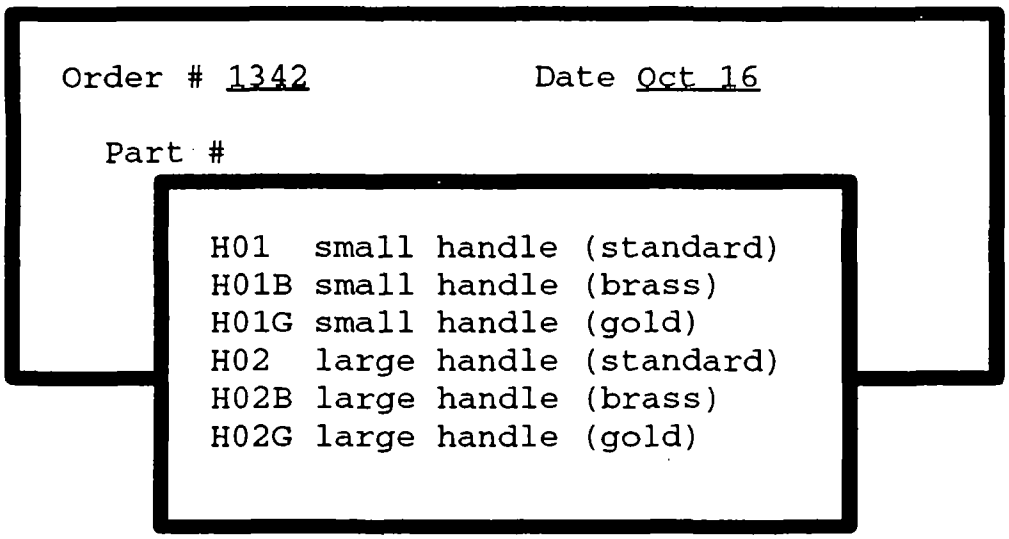


Figure 32: Item Selection Screen Example

When an item selection screen is filled dynamically, consider using the **JAM** library function `sm_svscreen` to avoid repeating the effort required to determine the entries. You can shrink the screen to fit the number of selections by calling the library function `sm_shrink_to_fit`. See the *Programmer's Guide* for details.

Starting with **JAM** release 5, function keys work on item selection screens.

Table Lookup

The table lookup entry serves a purpose very closely related to item selection. In the `table lookup` entry, you also specify the name of a screen, but the screen is not displayed at runtime. Instead, its contents are used to validate the entry in the field.

Commonly, an item selection screen is also used for table lookup (in fact, they are created in the same fashion). This ensures that the list of entries on the item selection screen (the fields with the menu edit) are the only entries permitted in the field. Even though the table lookup screen is not displayed, the screen is otherwise processed like a normal screen: the screen binary is re–read and screen entry and exit functions are called. This can cause significant redundant overhead, particularly if a database query is used to populate the screen. In such a case, or when the same table lookup is to be used repeatedly, consider using the **JAM** library function `sm_svscreen`. See the *Programmer's Guide* for details.

To specify a table lookup screen, enter the name of the screen in the `table lookup` field. For example, the following entry specifies that the `states.itm` screen is to be used for table lookup:

```
table lookup: states.itm
```

Note that the table lookup is done only for non–blank fields. Specify the 'data required' field edit if the field cannot be blank.

Status Text

Status text attached to a field is displayed on the status line whenever the cursor is in that field. You can embed display attribute and keytop codes in the status text that will be translated at run time. These codes are briefly described below; for a more complete reference, please see the documentation for the function `sm_d_msg_line` in the *Programmer's Guide.*

- If a string of the form `%A`***nnnn*** appears anywhere in the status text, where ***nnnn*** is a four digit hexadecimal number; the corresponding display attribute will be applied to the remainder of the text, or until another `%A` is found. To combine attributes, add the numbers together in *hexadecimal.* The possible values for ***nnnn*** are shown below.

Attribute	*Hex Code*	*Attribute*	*Hex Code*
Non Display	0008	Blinking	0040
Reverse Video	0010	Highlighted	0080
Underlined	0020	Low Intensity	1000
Foreground Colors		Background Colors	
Black	0000	Black	0000
Blue	0001	Blue	0100
Green	0002	Green	0200
Cyan	0003	Cyan	0300
Red	0004	Red	0400
Magenta	0005	Magenta	0500
Yellow	0006	Yellow	0600
White	0007	White	0700

- If a string of the form `%K`***kkkk*** appears anywhere in the message, ***kkkk*** is interpreted as the short name of a **JAM** logical key (e.g `XMIT`, `EXIT`, `PF1`, etc.). See section 2.3 of the Keyboard Entry chapter for a list of logical key names. If that key has a keytop defined in the key translation file for the keyboard being used, that label will replace the `%K` and the name. For example, `%KEXIT` is replaced with `Esc` when using the JYACC–provided keyboard file for the IBM PC. This enables keystroke instructions using the labels on the keys of the real keyboard. If there is no keytop, the `%K` is stripped out and the name remains.
- If the status text begins with a `%B`, **JAM** will issue a beep on the terminal.

Memo Text

Attaching memo text to a field is a feature provided by **JAM** to allow authors to attach information to a field that is not provided by any of **JAM**'s edits. **JAM** makes no use of the information in the memo text fields, but programmers can inspect the text and take action on it using the `sm_edit_ptr` library routine. Alternatively, comments can be stored in the memo text field. Nine lines of memo text are available. Since the field is scrollable, the ZOOM key can be used to view all nine lines at once.

Prior to JAM release 5, memo text was commonly used to pass parameters to field and screen functions. This need is greatly reduced by the ability to prototype functions. Prototyped functions can access hook string arguments; this was not possible for field and screen functions prior to release 5. The *Programmer's Guide* contains a discussion of function prototyping.

Miscellaneous Edits

Miscellaneous edits are specified in the Miscellaneous Edits Menu shown below.

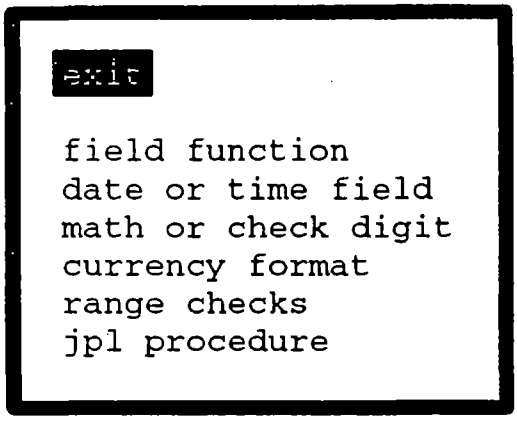


Figure 33: Miscellaneous Edits Menu

The available miscellaneous edits are described in the sections that follow. Briefly, the edits are:

- `field function`
  Field entry, exit, and validation functions written in C or JPL.
- `date or time field`
  Date and time field edits.
- `math or check digit`
  Math and check digit calculations.
- `currency format`
  The currency symbol, decimal symbol, decimal places, thousands separator, and other edits can be specified.
- `range checks`
  Up to 9 separate numeric or lexicographic ranges can be specified.
- `JPL procedure`
  A complete JPL procedure, including JPL subroutines, can be entered here. This procedure is called by JAM as part of the field validation process.

Field Functions

Field functions, selected from the Miscellaneous Edits menu, are hook functions written in C or JPL that are called by **JAM** at the field entry, exit, and validation hooks (The hook is where the function is attached). The function name, and optionally the arguments, are specified by entering a hook string into the Field Function Screen shown below.

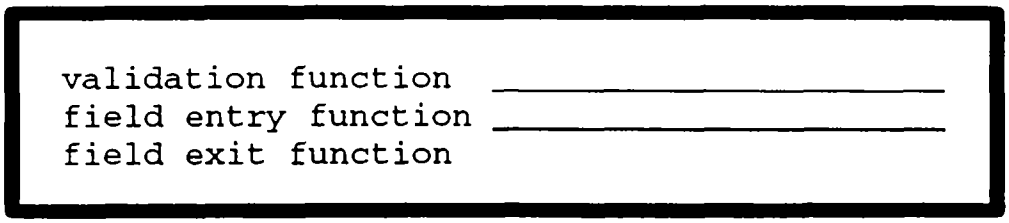


Figure 34: Field Function Screen

The format of a hook string that calls a C function is:

cfuncname {*arg* ...}

For example, the following hook strings call C functions:

```
hilight
chgcolor red
chgcolor :color
```

The format of a hook string that calls a JPL function is:

```
jpl jplfuncname {arg ...}
```

For example, the following hook strings call JPL functions:

```
jpl lookup
jpl chkmin 1000
jpl chkmin :minval
```

Note that, unlike control strings, hook strings that call functions *do not* start with a caret(^).

You must prototype screen, field, and group hook functions in order to pass them arguments. Colon preprocessing is performed on the arguments before the arguments are parsed. JPL will not execute a field function if the control line includes an argument. (A JPL function may use the "atch" command to pass an argument to an installed field function.) If the hook function is not prototyped, then it is passed the field number, occurrence number, contents, and a flag indicating the type of hook (entry, exit, or validation). Please see the *Programmer's Guide* for details on installing hook functions, prototyping functions, and hook function arguments. Colon preprocessing is discussed in the Authoring Reference chapter (section 7.1).

A field entry function is called when the cursor enters a field. A field exit function is called when the cursor leaves a field. **JAM** guarantees that a field exit function will be called exactly once for each time the corresponding field entry function is called. Therefore, a field exit function is called when the cursor is on a field in a screen that is de–activated[11]. Correspondingly, a field entry function is called for the field that the cursor is positioned upon when a screen is activated[12]. The *Programmer's Guide* explains how to install default field entry and exit functions on an application–wide basis.

A field validation function is called whenever a field is validated. By default, this occurs when the field is tabbed from (with TAB or via auto–tab) and when the XMIT key is pressed. For fields that are members of menus, radio buttons, or checklists, the validation function is not called as part of validation. The validation function for such fields is called instead when that field is selected. For checklist fields, the field validation function is also called when a field is deselected.

Prior to **JAM** release 5, field validation was often used to do field exit processing; field exit functions should now be used since exiting a field does not always cause field validation to be performed (e.g. when EXIT is pressed, when a field is exited via an arrow key). See the **JAM** library function `sm_option` in order to change the circumstances under which field validation occurs.

Date And Time Field

A date or time edit. selected from the Miscellaneous Edits Menu, can be used to enforce a format for entered dates or times, or to determine the format in which the system date or time is displayed. A date/time format string consists of literal characters and substitution variables. A literal character forces entry of that particular character. A substitution variable forces entry of a string that matches the format specified by the substitution variable. For example, the format string `HR:MIN:SEC` would accept `09:19:21`. The Date/Time Field Screen is shown below.

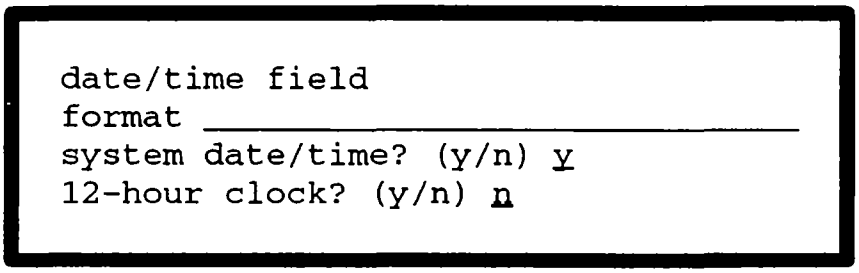


Figure 35: Date/Time Field Screen

Enter the format string in the `format field`. To specify that the field should be initialized with the system date/time, type a `y` in the system `date/time? (y/n)` field.

11. Screen de–activation occurs when an active screen is closed or hidden.

12. Screen activation occurs when a new screen is opened or when a de–activated screen is exposed.

To specify a 12 hour clock, type a `y` in the `12-hour clock? (y/n)` field. Type an `n` for a 24 hour clock.

A system date/time field is updated when the screen is opened, when the FIELD ERASE key is pressed while the cursor is on the field, when the CLEAR key is pressed (FIELD ERASE and CLEAR will work only if the field is not protected from clearing), or when a new array occurrence is allocated.

The substitution variables for date/time format strings are shown in the table below.

Substitution Variable	*User Input*	*Substitution Variable*	*User Input*
YR4	4 digit year	DAYL	long day name
YR2	2 digit year	DAYA	abbreviated day name
MON	month number	HR	hour
MON2	month number, zero filled	HR2	hour, zero filled
MONL	long month name	MIN	minute
MONA	abbreviated month name	MIN2	minute, zero filled
DATE	day of month	SEC	second
DATE2	day of month, zero filled	SEC2	second, zero filled
YDAY	day of year	AMPM	am or pm
DEFAULT DATE	MON/DATE/YR2	DEFAULT TIME	HR:MIN2
		DEFAULT	MON/DATE/YR2 HR:MIN2
Substitution variables must be entered in upper case!			

Figure 36: Substitution Variables for Date and Time.

The following table contains several sample format strings.

Format String	*Example of Acceptable Input*
MON2/DATE2/YR2	03/03/89
DATE2/MON2/YR4	19/09/1956
DAYL MONL DATE, YR4	Saturday December 26, 1954
HR:MIN2AMPM	2:04PM
DEFAULT	3/3/89 14:04

Figure 37: Example Date/Time Format Strings

To assist development standardization, a developer can define a substitution variable to take the place of an entire format string. For example, the variable STDDATE could be defined to be an installation–standard date format. This can be useful in applications that may be used with multiple languages where different date formatting conventions are used. Please refer to the *Configuration Guide* for more information.

Math or Check Digit

Math and check digit calculations, selected from the Miscellaneous Edits Menu, can be attached to a field. The calculations are performed when the field is validated. Math calculations can access and change any field, group, or LDB entry. A check digit calculation is used to validate a digits–only field according to a standard check digit algorithm. The mod–10 and mod–11 algorithms are supported[13]. The Math and Check Digit Screen is shown below.

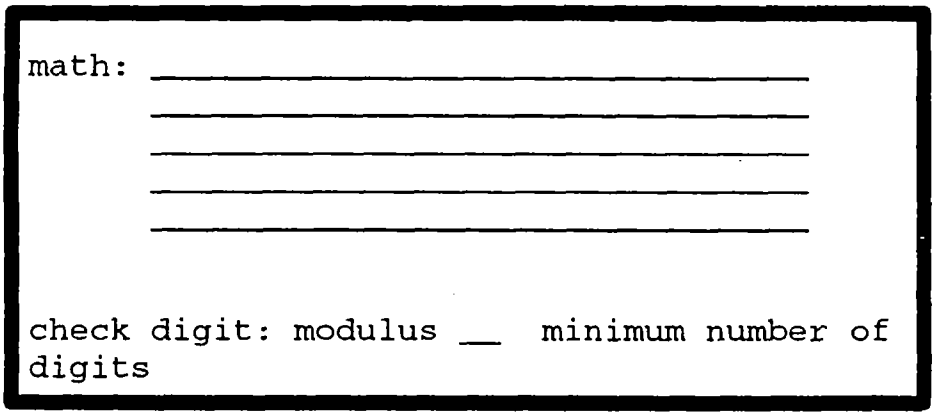


Figure 38: Math and Check Digit Screen

13. The source code to the check digit validation function `sm_ckdigit` is provided with the JAM library, and may be modified to support other check digit algorithms.

You can attach one check digit calculation and one or more math calculations to a field. Math calculations are separated by semicolons, even if they are on different lines.

A math expression starts with an optional floating point size specification for the destination field. This specification has the form `%`*m*`.`*n* or `%t`*m*`.`*n* where *m* specifies the total number of characters in the output and *n* the number of digits after the decimal point. The `t` forces truncation, rather than rounding, of the result. If no size is supplied, the total length defaults to the length of the destination field or LDB entry. The number of decimal places defaults to the number of decimal places in the currency edit attached to the destination field; if there is no currency edit, it defaults to the precision given in the float, double, zoned, or packed data type edit attached to the destination field; if there is no data type edit, it defaults to two decimal places.

The optional size specification is followed by the destination field designation (e.g. field name or number), an equal sign, and the body of the expression. The expression body can contain numeric constants, field designations, parentheses, and the arithmetic operations +, −, *, /, and ^ (raise to a power).

Fields can be designated by name or by number. Field numbers must be preceded by a # sign. For example, `#12` refers to field number 12. You can use relative referencing to refer to a field relative to the field number of the current field. Relative references must be preceded by a plus or a minus sign. The field just before the current field is `#-1`. The fifth field following the current field is `#+5`. To refer to the current field, use `#+0` or `#-0`.You can designate a particular array occurrence by appending the occurrence number in square brackets to the field designation. Again, the number may be absolute or relative. If a designated field is part of an array, *and* the designation contains no occurrence subscript, then the current occurrence number is used as the array index. The current occurrence number is the occurrence number of the data in the field being processed. If the current occurrence number is greater than the number of occurrences in the designated array, an error results.

Typical math expressions look like this:

```
%8.0 #3 = #1 * 12 + 2

flda[2] = (flda[1] - 6.235) / fldb[1]
```

As an example of referring to the current field, consider the following two math expressions:

```
#-3 = #+0 * #+3

#-3 = #+0+6
```

In the first expression, the field that occurs three fields before the current field is set to the value obtained by multiplying the value in the current field by the value in the third field after the current field. In the second expression, the field three fields before the current field is set to the sum of the value of the current field and the integer six.

There are two special functions available in math expressions: `@sum` and `@date`. `@sum` yields the sum of all the occurrences in a given array:

```
@sum(array1)  Total values in array1.
@sum(#2)  Total values in the array containing field 2.
```

`@date` yields the number of days between 1/1/1753 and the argument to `@date`. The argument must be either a field (name or number) or LDB entry (name) with a date edit (or in `MON/DATE/YR2` or `MON/DATE/YR4` format), or a literal date in the format specified in the message file under the heading `SM_CALC_DATE`. The default literal format is `%m%d%4y` (`MON/DATE/YR4`). The following are each valid examples:

```
@date(quarterday)
@date(#-1)
@date(3/31/1985)
```

An error occurs if the specified field does not have the proper format. The number resulting from the calculation is interpreted as the number of days elapsed since 1/1/1753. If a destination field is a date field, then the result is displayed according to the date field's format; otherwise the date is displayed as the number of days since 1/1/1753. If `field1` and `field2` are both date fields,

```
field2 = @date(field1) + 30
```

will set the date in `field2` to 30 days past the date in `field1`.

The following expression sets `daysgoneby` to the number of days between `field1` and `field2`:

```
daysgoneby = @date(field2) - @date(field1)
```

Currency Format

The currency format edit, selected from the Miscellaneous Edits Menu, is used to designate a field to hold monetary values. Numeric data is formatted and displayed as a monetary amount. The formatting occurs during field validation. Non–numeric data entered into a currency field is discarded. Therefore, it is usually less confusing to the user if the field also has the clear on input field edit.

See the *Programmer's Guide* for a fuller discussion of internationalization considerations.

The Currency Format Screen is shown below.

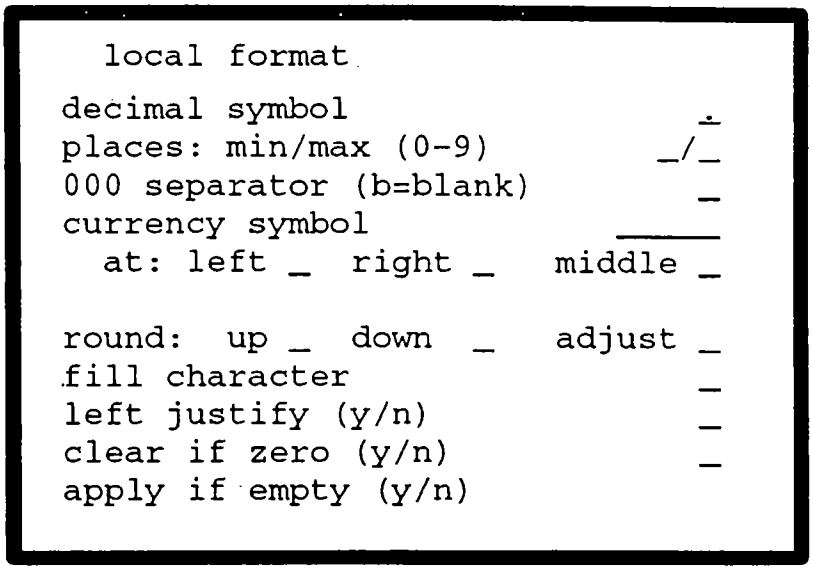


Figure 39: Currency Format Screen

Fill in the screen, using the edits described in the following list:

- `local format`

  The name of a local format defined in the message file. It is case sensitive. Local formats are stored in the message file so that the same application running in two different countries can use identical screens, but the currency fields would be formatted according to local custom and monetary units. The local format establishes the values for currency formatting: decimal symbol, places, 000 separator, currency symbol, and currency symbol position. When a local format is specified, the previously listed format entries cannot be changed. Additional information is given below.

- `decimal symbol`

  The character used as the decimal separator (radix separator). Any single character may be used.[14]

- places: min/max

  The minimum number of decimal places followed by the maximum number of decimal places. In both cases, these must be numbers between 0 and 9. If none are entered, the minimum defaults to 0 and the maximum to nine.

14. The decimal symbol defaults to the value of SM_DECIMAL in the JAM message file (usually a single dot). If SM_DECIMAL is not in the message file, then JAM tries to determine the decimal symbol by asking the operating system. Some operating systems do not support this feature, For those that do, the mechanism for specifying the decimal symbol varies from operating system to operating system. See the JAM Installation Guide for your operating system.

- `000 separator`

  A single character thousands separator. Use `b` to indicate a blank space. If none is chosen, no thousands separation will occur.

- `currency symbol`

  A one to five character currency symbol. Because `currency symbol` is a regular **JAM** field, it is not possible to enter trailing spaces, as they are stripped off. To specify a leading currency symbol separated from the data by a space (e.g. `FF 123.456,78`) use a period to indicate the space. If no symbol is entered, no symbol will be used to denote currency. The currency symbol may occur to the left (the default) or right of the amount, or in place of the decimal point. Place a `y` by the left, right, or middle entry to specify currency symbol placement. If middle, the currency symbol overrides the decimal symbol.

- `decimal round`

  The amount may be rounded up, rounded down, or adjusted by standard rounding (i.e. round up above .5) to the number of decimal places specified (the default). Enter a `y` by one of the three rounding options. The following table shows the effect of rounding on several example numbers. Note the impact that rounding has on negative numbers.

Number Entered	*Round Up*	*Round Down*	*Adjust*
2.056	2.06	2.05	2.06
–2.056	–2.05	–2.06	–2.06
2.055	2.06	2.05	2.06

Figure 40: Rounding to 2 Decimal Places (Maximum)

- `fill character`

  A character that will replace any blank after formatting.

- `left justify`

  By default, a currency field is right justified. Enter a `y` here if you want it left justified. Note that this justification occurs after validation. If you want the user to begin entering data in the rightmost position in the field, assign the field the right justify field edit (see page 40).

- `clear if zero`

  Enter a `y` here if the field is to be cleared when validated if the amount in the field is zero. For example, suppose the field is eight characters long and right–justified with the asterisk as the fill character. If the field is 0 and not cleared, then it will display as:

```
*******0
```

- `apply if empty`

  Enter a `y` here if the format is to be applied even when the field is blank (ie. — the operator TABs out of the field), in which case **JAM** displays a formatted zero (e.g. `$0.00`). This has no effect if `clear if zero` is specified.

The possible local format names are defined by the message file entries `FM_0MN_CURRDEF` through `FM_9MN_CURRDEF`. The corresponding formats are defined by the message file entries `SM_0DEF_CURR` through `SM_9DEF_CURR`. The table below shows the default message file entries.

Message File Entry for Local Format Name	*Local Format Name (enter into local format field)*	*Message File Entry for Local Format Specification*	*Local Format Specification*
FM_0MN_CURRDEF	CURRENCY	SM_0DEF_CURR	.22,1$
FM_1MN_CURRDEF	PLAIN	SM_1DEF_CURR	.09,
FM_2MN_CURRDEF	NUMERIC	SM_2DEF_CURR	.09
FM_3MN_CURRDEF	DEFAULT3	SM_3DEF_CURR	.09
FM_4MN_CURRDEF	DEFAULT4	SM_4DEF_CURR	.09
FM_5MN_CURRDEF	DEFAULT5	SM_5DEF_CURR	.09
FM_6MN_CURRDEF	DEFAULT6	SM_6DEF_CURR	.09
FM_7MN_CURRDEF	DEFAULT7	SM_7DEF_CURR	.09
FM_8MN_CURRDEF	DEFAULT8	SM_8DEF_CURR	.09
FM_9MN_CURRDEF	DEFAULT9	SM_9DEF_CURR	.09

Figure 41: Default Message File Entries for Local Currency Formatting

Each local format specification string is of the form ***rmxtpccccc***, where:

r	decimal symbol, usually ., ,, or b (blank).
m	minimum number of decimal places

x	maximum number of decimal places
t	thousands separator, usually ., ,, or b (blank).
p	placement of currency symbol: l=left, r=right, m=middle.
ccccc	up to 5 characters for the currency symbol.

Given the default message file, the following table shows how several strings would be formatted according to the CURRENCY, NUMERIC, and PLAIN local formats:

Entered Data	*Formatted Data*		
	CURRENCY	NUMERIC	PLAIN
12345	$12,345.00	12,345	12345
1234	$1,234.00	1,234	1234
123.45	$123.45	123.45	123.45
123.456	$123.46	123.456	123.456

Figure 42: Local Currency Formatting Examples

Range Checks

This option, selected from the Miscellaneous Edits Menu, allows specification of up to nine ranges of minimum and maximum values for a field. If the field contains a digits–only or numeric character edit, the values entered will be compared numerically. In all other cases, the values will be compared as character strings.

If, for a given range check pair, only the lower bound is specified, then all field data greater than or equal to that lower bound are accepted. Correspondingly, if only the upper bound is specified, then all field data less than or equal to that upper bound are accepted. Empty fields are always considered in range.

The Range Check Screen is shown below.

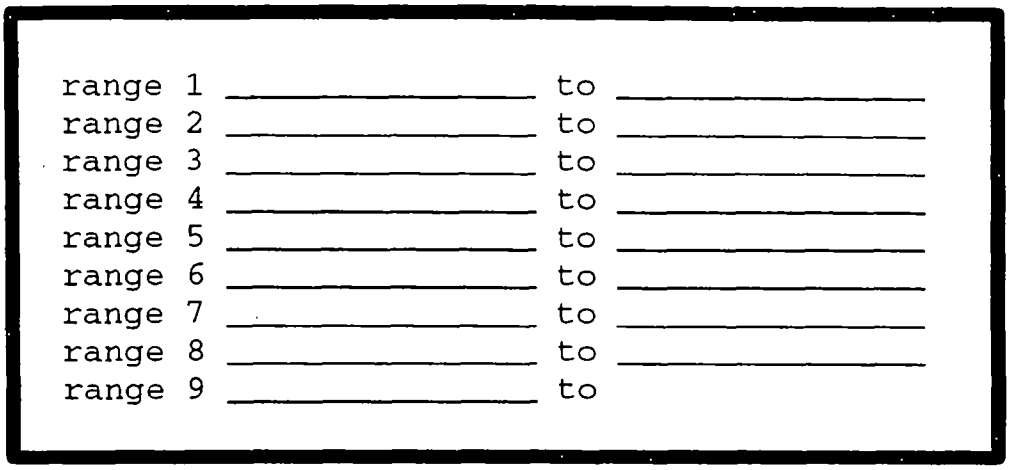


Figure 43: Range Check Screen

JPL Procedure

A complete JPL procedure, including JPL subroutines, can be entered here. This field–level JPL procedure is called by **JAM** as part of the field validation process, just after the field validation function. Refer to the *JPL Guide* for instructions on writing JPL procedures.

To enter JPL code, choose the `jpl procedure` option from the Miscellaneous Edits menu. The JPL Procedure Screen will appear. See the section on screen–level JPL, page 32, for a discussion of this screen.

The JPL code entered here can be called only when the field is validated, and only from here; it cannot be called by the field validation function. Store JPL in the screen–level JPL Procedures Screen (or in a file) if the procedures must be called from field functions, control strings, or from within JPL procedures for other fields. The advantages of storing JPL procedures in a field–level JPL Procedures Screen are:

- The JPL is partially syntax–checked and compiled when the JPL Procedures Screen is closed.
- The JPL is stored with the screen, rather than in a separate file.
- The JPL code can be copied from screen to screen, via the data dictionary, for use by the same field on different screens.

Field Size

A field must occupy at least one position on the screen. However, **JAM** fields are virtual in the sense that the capacity of a field can be larger, in two dimensions, than its ons-

creen capacity: a field can shift and scroll. Shifting extends a field's width beyond its onscreen width. Scrolling permits a field to hold more data items than will fit onscreen.

Each **JAM** field is part of an array, even if it is the only field in the array. The first array field (usually the field painted with the draw field symbol) is called the *base field.* Additional fields of the same array are created by specifying that the field has more than one *element*; the base field is the first element of the array. Each element has a unique field number, but shares every other characteristic with the other elements of the array. The array slots that can contain data are called *occurrences.* The elements of an array are the mechanism through which the array's occurrences are viewed (much like a viewport permits the entire screen to be viewed in sections). A *scrolling array* can have more occurrences than elements. A *simple array* has the same number of occurrences as elements.

Array occurrences can be allocated up to the maximum number of occurrences specified for the array. See the Scrolling Arrays section (page 144) of the Authoring Reference chapter for a discussion of when occurrences are allocated. The number of allocated occurrences is never less than the number of array elements. The number of the highest numbered non–blank allocated occurrence in the array is called the number of populated occurrences. Programmer's note: the library functions `sm_max_occurs` and `sm_num_occurs` return the maximum and populated number of occurrences respectively.

Scrolling arrays can be synchronized so that they scroll together. This helps manage related information in table–oriented screens. Any set of scrollable arrays on a screen can be synchronized, provided that they have the same number of onscreen elements and the same maximum number of occurrences. Manual synchronization of arrays is discussed later in this chapter (see SPF7 on page 95). **JAM** will automatically synchronize two or more arrays if they are *parallel.* Parallel arrays meet the following criteria:

- They have the same number of onscreen elements.
- They have the same maximum number of occurrences.
- If vertical, all base fields must start on the same row, and the offset between elements must be the same.
- If horizontal, all base fields must start on the same column, and the sum of onscreen field length and distance between elements must be the same.

To recap, a field's size has four parameters:

- The onscreen length.
- The maximum length.
- The number of array elements.

- The maximum number of array occurrences.

To modify the field size parameters, choose size on the Field Characteristics menu. The Field Size Screen will appear as shown below. It contains the field's current size parameters.

```
                Onscreen Information
Length:  __
Number of elements: ___
Distance between elements:___ Horizontal?_ Word wrap?n

Offscreen Information

  Maximum shifting length: ___  Increment: ___
  Number of occurrences:  ____  Page size: ___
  Circular? _  Isolate? _

Alternative scrolling method:
```

Figure 44: Field Size Screen

The following information can be modified:

- `Length`
  The onscreen length of the field. The field can not overlap another field or extend into the screen border.
- `Number of Elements`
  The number of onscreen field elements in the array. Each element has its own field number.
- `Distance between Elements`
  The number of columns between successive horizontal array elements or one greater than the number of rows between successive vertical array elements (ie. — if the distance = 1, then the elements will appear on successive lines). The default value is 1.
- `Horizontal?`
  For a vertical array (elements above and below each other in a column) enter `n` or leave blank. For a horizontal array (elements stretching out right and left of each other) enter `y`.
- `Word wrap?`
  Enter `y` so that words will shift between occurrences to avoid being split,

and to avoid large unused portions of occurrences. This is useful for blocks of text in vertical arrays.

- `Maximum shifting length`
  The maximum length of data to be put in the field. Must be greater than the onscreen length to make the field shiftable.

- `Increment`
  The number of characters by which the field data should be shifted when the user moves the cursor beyond the onscreen field edge.

- `Number of occurrences`
  The maximum number of occurrences to be put in the array. This must be greater than or equal to the number of onscreen elements to make a scrollable array. If no number is entered, then the array is a simple array. **Simple arrays cannot be synchronized.**

- `Page size`
  The number of occurrences by which a PAGE UP or PAGE DOWN keystroke should scroll the field. For the default, leave blank. This must be less than or equal to the number of elements. By default, this quantity is one less than the number of elements, or one for a scrolling field with one element.

- `Circular?`
  Enter `y` if the scrolling algorithm should wrap around from the last occurrence of a scrolling array to the first, or from the first to the last, when the down/up arrow keys (that would normally leave the array) are pressed. Note that all array occurrences must be allocated for the array to scroll circularly.

- `Isolate?`
  Enter `y` if the array is parallel to one or more other arrays and you do not want it to be automatically synchronized with them.

- `Alternative scrolling method`
  A hook string to invoke an alternative scroll driver to handle scrolling for the array. This can be used, for example, to apply the optional disk–based scrolling technique supplied with **JAM** in order to save memory. You can also write your own scroll driver and install it as the default driver in order to control the scrolling process. See the *Programmer's Guide* for more information.

Data Type

The data type is used by the `f2struct` utility to construct C language data structures. These structures are used by certain **JAM** library functions (e.g. `sm_rdstruct` and

sm_wrtstruct) to exchange data between **JAM** and C programs. In addition, the sm_ftype library routine examines the data type edit when determining a field's major data type. The Data Type Menu is shown below.

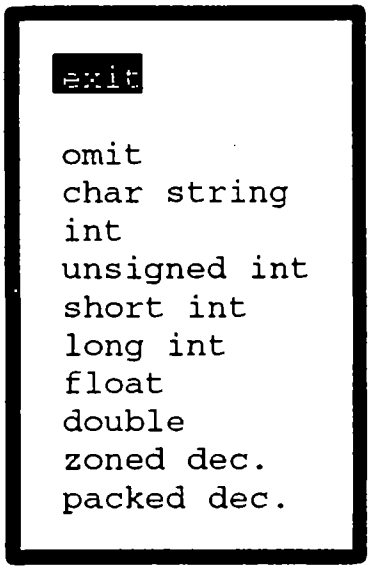


Figure 45: Data Type Menu

The special type omit means that the field should be omitted from the data structure. The actual labels used for generating the data structures are taken from the message file, where they may be adapted to suit your programming language. The default message file comes with type labels suitable for C language programming. In addition, zoned and packed decimal data types are provided.

The data types int, unsigned int, short int, long int, float, and double affect colon–plus processing in **JAM**/Db*i*. Refer to the **JAM**/Db*i* manual for details.

If you select float, double, zoned dec, or packed dec then the Data Type Precision Screen is displayed as shown below.

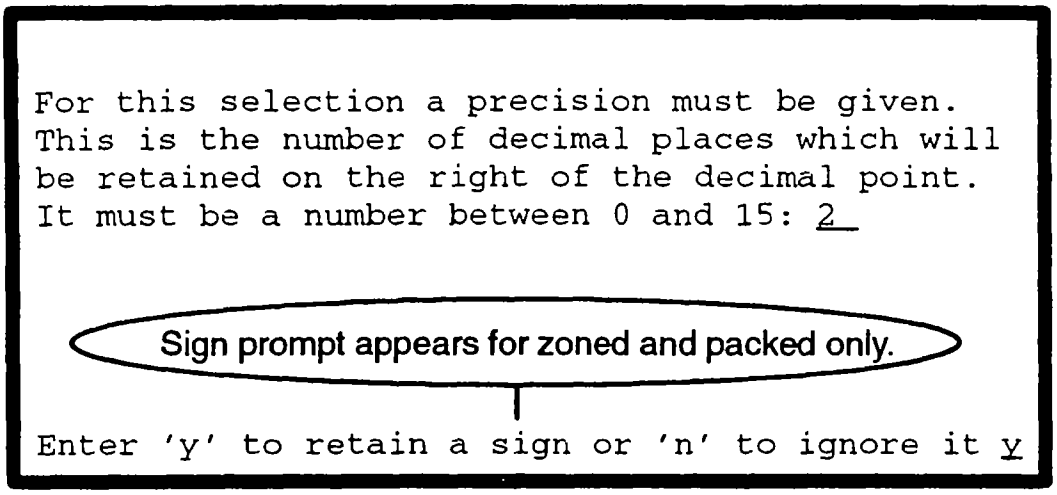


Figure 46: Data Type Precision Screen

The default precision is the number of decimal places in the currency edit, if any, or 2 otherwise. It is used by `sm_rdstruct` and `sm_wrtstruct` to determine the number of decimal places to preserve. It is also used in math calculations, in the absence of a currency edit, to determine the number of decimal places to display.

The `packed dec` and `zoned dec` types may either be signed or unsigned. When either `packed dec` or `zoned dec` are specified, the Data Type Precision Screen displays a sign retention prompt. Enter `y` for signed, or `n` for unsigned. Sign retention is used by `f2struct` and `dd2struct` to determine the size of fields in the data structure it creates, and by `sm_rdstruct` and `sm_wrtstruct` to determine how to transfer data between the structure and the field. It does not impact math calculations.

See the *Programmer's Guide* (`sm_rdstruct` and `sm_wrtstruct`) and *Utilities Guide* (`dd2struct`) for more information.

4.3.4

Field Summary (PF5)

Viewing and editing field characteristics by wading through the entire PF4 menu tree can be tedious; the Field Summary Screen is provided for viewing and changing frequently modified field characteristics. To display the Field Summary Screen, position the cursor in a field and press PF5. The Field Summary Screen appears as shown below.

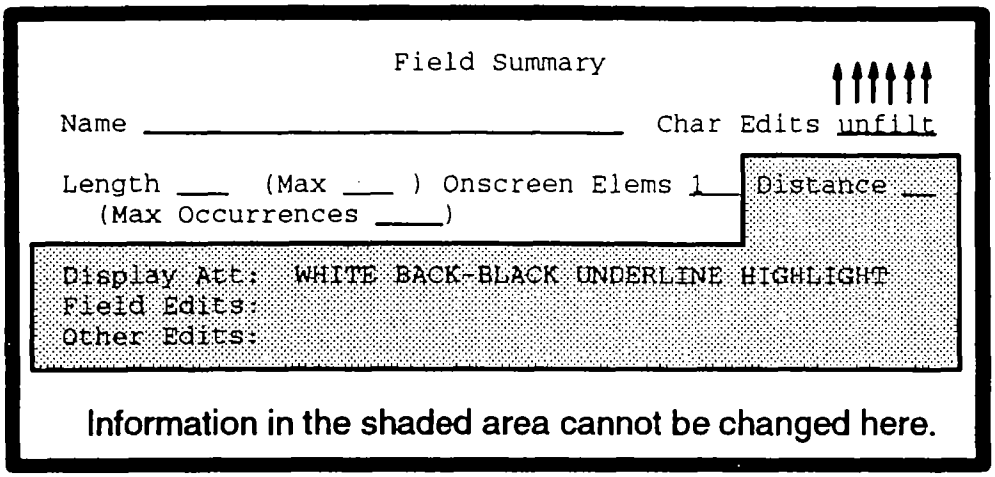


Figure 47: Field Summary Screen (Note: for readability, the screen shown here is not exactly as it appears in **JAM**.)

While the Field Summary Screen is active, you can press PF4 to enter the PF4 menu hierarchy. **Beware: any changes made while navigating that hierarchy will be lost if the Field Summary Screen is closed by typing EXIT; use XMIT to save your changes.**

Each field characteristic is fully described elsewhere in this chapter, but we will briefly discuss how to make changes to the characteristics listed in the Field Summary Screen. Note that some characteristics can be viewed here, but not changed.

- `Name`

  Field name. It can be changed by typing into the field.

- `Char Edits`

  This is a circularly scrolling array with one element displaying the field's character edit. To change the edit, Use the up and down arrow keys to select the one you want. The one that is visible when you hit XMIT on the window will be used.

- `Length`

  The field's onscreen length. It can be changed by typing into the field.

- `(Max)`

  The field's maximum shifting length. It can be changed by typing into the field.

- `Onscreen Elems`

  The number of elements (onscreen fields) in the array. It can be changed by typing into the field.

- `Distance`

  The distance from one array element to the next. It cannot be changed directly. To modify, press PF4 and enter the regular menu hierarchy.

- `(Max Occurrences)`
  The maximum number of occurrences in the array. It can be changed by typing into the field.
- `Display Att`
  The field's display attributes. These can not be modified directly. To modify, press PF4 and enter the regular menu hierarchy.
- `Field Edits`
  The field's edits. These can not be modified directly, use the PF4 key.
- `Other Edits`
  Other attachments and edits. Use the PF4 key to modify.

4.3.5

Select Mode for Editing (PF6)

Select mode is used to move and copy sets of fields and display areas within a screen and between screens. It is also used to delete fields and modify display attributes for sets of fields. A clipboard permits these screen components to be saved between editing sessions. To enter select mode, press the PF6 key when in draw or test mode. The status line will indicate select mode by displaying an `@` in the first position. The full status line is pictured below.

```
@F2clip F3box F4disp F5del F6sel F7move F8copy F9rept F10resel
```

Figure 48: Select Mode Status Line

Press the EXIT key to exit select mode and return to draw or test mode. As you can see from the status line pictured above, the function keys perform different functions in select mode than they do in draw or test mode.

In select mode, you define and operate on *select sets*. A select set is a set of display areas and/or fields which are operated on as a unit. You might think of a select set as being similar to a highlighted block of text in a word processor environment. Like such a block, you can copy or move or delete a select set, or save it to a file. In the following sections, we will discuss defining and operating on select sets. The function keys are summarized briefly below.

- `PF2 clip` Activate clipboard menu.
- `PF3 box` Draw box to define select set.
- `PF4 disp` Change display attributes of select set.

- `PF5 del` Delete select set. The set is copied to clipboard Z.
- `PF6 sel` Select or de–select an object.
- `PF7 move` Move select set.
- `PF8 copy` Copy select set.
- `PF9 rept` Repeat last operation.
- `PF10 resel` Reselect set from previous select mode session.

In select mode, the cursor behaves as if in draw mode; however, text cannot be entered.

Establishing Select Sets (PF6, PF3, PF10)

As mentioned above, select mode is entered by pressing PF6 when in draw or test mode. If the cursor is in a field or display area when PF6 is pressed, then that field or display is the initial member of the select set; otherwise the select set will be empty initially. Each field and display area in the select set is marked by highlighted square brackets([ ]).

The *Field Select* (PF6) key toggles a field or display area in and out of the select set. In other words, to add a field or a display area to the select set, position the cursor on it and press PF6. To deselect the field or display area, press PF6 again.

The *Box Select* (PF3) key selects the fields and display areas in a rectangular region of the screen. If part of a field is in the box, then the whole field will be selected (operations cannot be performed on parts of fields). If part of a display area is in the box, only that part will be selected.

To use Box Select, place the cursor in one of the four corners of the region you wish to select. Press the PF3 key, which places a highlighted asterisk (*) at the cursor position. This is called the anchor position. Now move the cursor to the opposite corner of the box. You will see asterisks appear at the other corners of the box as you move the cursor, but the original anchor asterisk will not move. Press PF3 again to add everything inside the box to the select set (XMIT, PF4, PF5, PF7, and PF8 also work). To abort the operation, press EXIT.

The *Reselect* (PF10) key, re–selects everything that was selected last time you left select mode. Re–select is useful for performing a different operation on the same set of items. Note that when you press PF10 the first time you enter select mode in a screen editing session, nothing will happen. This is because there is nothing to re–select.

Select Mode Operations (PF4, PF5, PF7, PF8)

The *Display* (PF4) key is used to set the display attributes for all the fields and display areas in a select set. Pressing PF4 causes the Display Attributes Screen (see page 24) to

display. The members of a select set can have different display attributes. Only those attributes changed from the Display Attributes Screen will be changed for the members of the select set. For example, if two fields with different attributes are selected (say one is highlighted and the other blinking) and the attributes window is used to assign a color of blue, only the color will be inherited when the attributes window is closed. The fields will still maintain all of their other unique attributes. To set an attribute to the default value that comes up when the attributes screen is opened, you must tab to that attribute and actively set it. So, for example, to set the foreground color of the select set to white, tab to the color field and select white. See page 24 for a discussion of the Display Attributes Screen.

The *Delete* (PF5) key deletes all fields and display areas in the select set. The deleted items are moved to clipboard Z. If you delete something by mistake, it can be restored from clipboard Z as discussed later in this chapter. Note that only the last deleted set may be "undeleted".

The *Move* (PF7) key moves a select set around the screen. To move a select set, press PF7. The contents of the selected items will vanish, leaving only the highlighted brackets that define the select set. Use the arrow keys to re–position the select set. Press PF7 again to complete the move. The *Copy* (PF8) key works the same way except that the select set is copied to a new location rather than moved.

A move or copy is rejected if it would result in overlapping fields. Overlapped display text is replaced by the overlapping field or display text. Display text that overlaps a field becomes initial data for that field. Note that newly created fields inherit all field characteristics of the copied field except the field name. This is because field names must be unique on a screen.

If a copied field is part of a group, then the newly created field becomes part of the same group.

The Repeat (PF9) key applies the most recent operation to another, or to the same, select set.

The Clipboard (PF2)

The clipboard enables copying select sets from screen to screen. For example, you can use the clipboard to make all name and address blocks, including descriptive text, on all screens of your application look the same.

By convention, we speak of "the clipboard". In reality, **JAM** supports 26 clipboards, named A through Z. Clipboard Z is used by **JAM** to store the most recently deleted field, display area, or select set; the previous content of Z is overwritten. You can "undo" a deletion by retrieving the contents of clipboard Z.

To use the clipboard, press PF2 from select mode. The Clipboard Control Screen will display as shown below.

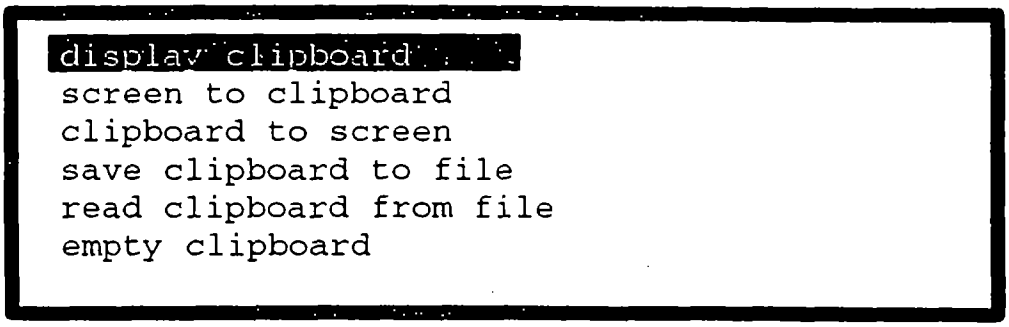


Figure 49: Clipboard Control Screen

This screen is a menu of clipboard commands; to execute one, type the first letter of the command or position the bounce bar over the command and press XMIT. You will be prompted for the clipboard name (a default will appear) and, if necessary, for a file name. The Clipboard Control Screen with prompts is shown below. After supplying the information, press XMIT to invoke the command.

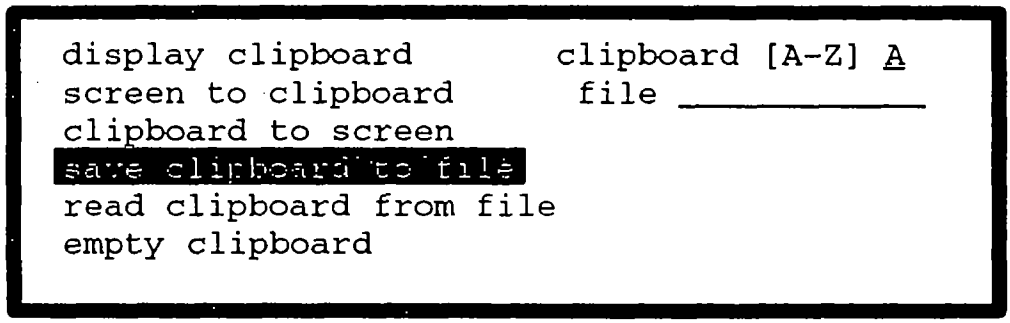


Figure 50: Clipboard Control Screen With Prompts

The Clipboard Control menu commands are explained below:

- `display clipboard`
  Show the content of the specified clipboard. Press space bar to return to the Clipboard menu.
- `screen to clipboard`
  Copy the select set to the specified clipboard. The clipboard name will default to the first empty clipboard.
- `clipboard to screen`
  Copy the content of the specified clipboard onto the screen. The Clipboard Control Screen is temporarily closed, and the contents of the clipboard are displayed at the cursor position on the edited screen. Use the arrow keys to move the clipboard contents around the screen as a unit. Press PF8 to complete the copy and return to the Clipboard menu (XMIT and PF7 also work). Press EXIT to abort.

- `save clipboard to file`
  Save the specified clipboard to the specified file. The clipboard is saved as a screen that has the same display attributes as the original screen, and the size of the smallest box that can contain the select set. This file can be opened as a **JAM** screen. Background attributes and draw field symbols are also retained, although if the clipboard is pasted onto an existing screen, they are lost.

- `read clipboard from file`
  Retrieve the clipboard content from a file created with the `save clipboard to file` command. Since clipboards are saved as **JAM** screens, any **JAM** screen can be retrieved in this fashion. When a screen is retrieved through the clipboard, its background color is lost and its border becomes multiple display areas.

- `empty clipboard`
  Delete the content of the specified clipboard.

Clipboard operations on select sets that include partial or complete groups will retain the group relationship. If a field that is part of a group is copied to the screen from the clipboard, then it is added to the group on the screen, if that group exists (i.e. if the name of the group in the clipboard is the same as the name of the group on the screen). If that group doesn't exist, then a new group is created with the attributes saved in the clipboard.

4.3.6

Simple Editing Commands (PF7, PF8)

The select mode move and copy operations, described above, are directly available in draw/test mode for single fields or display areas. To move or copy a field/area, move the cursor to that field/area and press PF7 or PF8 respectively. The field/area will be marked by highlighted square brackets. Move the field/area (or copy field/area) with the arrow keys to the desired location, and press PF7 or PF8 again.

4.3.7

Repeating Operations (PF9)

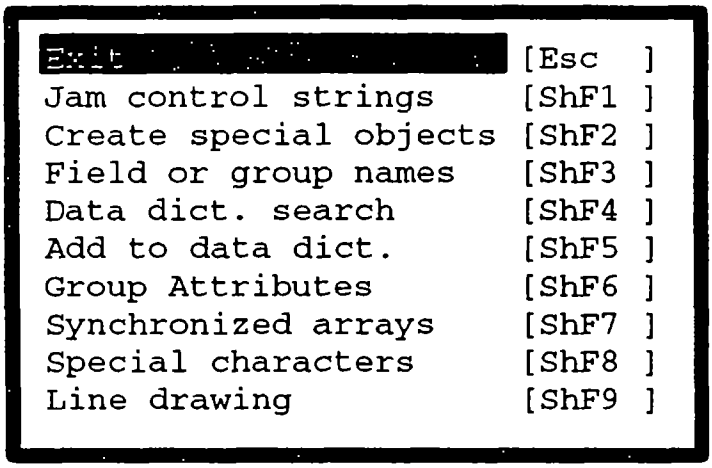
The select mode repeat operation, described above, is directly available in draw/test mode via the PF9 key. The last operation performed in draw/test mode is repeated when PF9 is pressed. The operations that can be repeated include: move, copy, graphics, and change field characteristics.

The repeat function works by recording and replaying keystrokes. This may produce surprising results in certain circumstances. For example, changing the field entry function name on a field, toggling from typeover to insert mode, and then moving the cursor to another field and pressing PF9 will result in inserting the new name in front of the existing name (as opposed to replacing the old name with the new name). Even if you remain in typeover mode, the replacement text must be longer than the replaced text (the safest method is to use FIELD ERASE before entering the new function name).

4.3.8

Shifted Function Key Menu(PF10)

The shifted function keys SPF1 through SPF10 also have meaning to the Screen Editor. They are not listed on the status line, but may be viewed by pressing the more key, PF10. This will display the Shifted Function Key Menu shown below.



The keytops for your keyboard will be displayed.

Figure 51: Shifted Function Key Menu

The Shifted Function Key options, which are explained in the following sections, can be accessed in one of three ways:

- Press the Shifted Function Key in draw or test mode. To do this, you need to know which key to press without any guidance on the display. This is the quickest method.
- Press PF10 in draw or test mode to display the Shifted Function Key Menu. The menu lists the shifted function keys; you can press the shifted function key itself while the menu is displayed, and the system will act exactly as if it had been pressed in draw or test mode.
- Press PF10 in draw or test mode to display the Shifted Function Key Menu. Select an option by positioning the bounce bar and pressing

XMIT or by typing the first letter of the option. The system will act exactly as if the key had been pressed in draw or test mode.

To keep the documentation simple, the shifted function key operations are discussed as if they are invoked merely by pressing the shifted function key in draw or test mode.

4.3.9

JAM Control Strings (SPF1)

Control strings (see the Overview and the Authoring Reference) direct the **JAM** Executive to display forms, display windows, call C and JPL programs, and invoke operating system programs. The Control String Screen, shown below, is used to associate control strings with function keys.

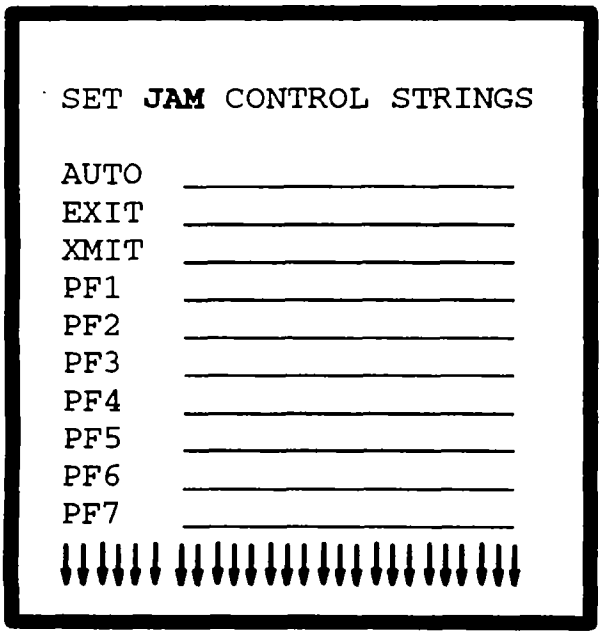


Figure 52: Control String Screen

A screen's Control String Screen contains a scrollable array of the control strings associated with function keys. This assignment of control strings to function keys applies only to the edited screen. The *Configuration Guide* explains how to make global assignments. See section 7.2 on page 124 for a detailed discussion of control strings.

The keys that can have control strings assigned to them are XMIT, EXIT, PF1 through PF24, SPF1 through SPF24, and APP1 through APP24. There is a special "key" named AUTO (there is no **JAM** logical key named AUTO) that can be assigned a control

string. To make an assignment, enter a control string on the line following the name of the key. If the control string is long, you may find the ZOOM key useful. When you enter a control string in the Control String Screen, it is stored with the screen, but takes up no space on the display.

For example, the following assignments cause XMIT to display the form `lookup`, PF1 to display the window `curstat`, PF2 to call the C function `sendmsg`, and PF3 to call the JPL function `chkerr`:

```
XMIT    lookup
PF1 &curstat
PF2 ^sendmsg
PF3 ^jpl chkerr
```

Hint: to enable a key to behave like EXIT, use the control string `^jm_exit`. To disable the EXIT key, use `^nop`. `^jm_exit` and `^nop` are built–in functions. See the *Programmer's Guide* for detail information on these and other built–in functions, or see section 7.2.4 on page 126 for a brief discussion.

The AUTO control string is invoked each time the screen is activated (i.e. when the screen is opened or exposed). The AUTO control string may be used to open a window which has an AUTO control string for opening another window and so on. This way a series of windows can be opened automatically. If AUTO control strings are used to open a series of windows, then closing the last window in the chain will cause them all to close.

4.3.10

Create Special Objects (SPF2)

The special objects are menus, groups (radio buttons and checklists), and screen name fields. SPF2 creates these objects quickly, using the most common attributes. Note that this is a shortcut—the objects may need to be modified (in some cases created from scratch using another method) to meet application requirements. See the Authoring Reference chapter for additional information on groups and menus. See SPF6 below (page 92) for information on assigning group attributes.

To create a menu, group, or screen name field, press SPF2 to display the Create Special Objects Screen shown below—then make the appropriate menu selection.

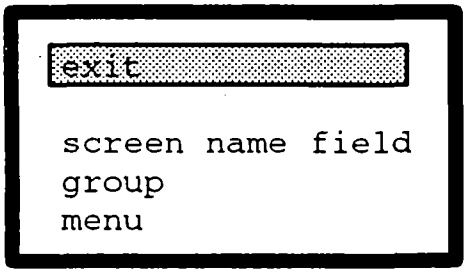


Figure 53: Create Special Objects Screen

Shortcut Menu Creation

Shortcut menu creation allows you to create a vertical or horizontal menu composed of two arrays. The first array contains the menu selection text. The second array contains the menu control strings. Control strings and menus are described in the Authoring Reference (sections 7.2 and 7.3 respectively).

Selecting `menu` from the Create Special Objects Screen will display the Menu Shortcut Screen shown below.

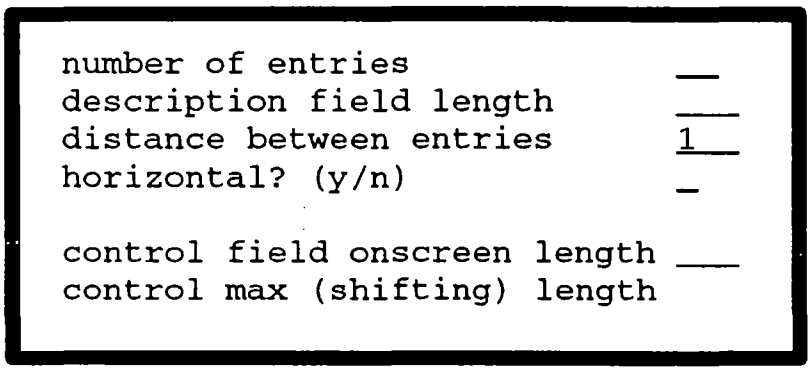


Figure 54: Menu Shortcut Screen

Complete the entries in the Menu Shortcut Screen as follows:

- `number of entries`
  The number of onscreen menu selections (number of elements in the menu selection array).
- `description field length`
  The length of the menu selection fields (they will contain the text of the menu selections).
- `distance between entries`
  The distance between menu entries, measured in columns for horizontal

menus and in rows for vertical menus. For horizontal menus, the distance is measured between the end of a menu control field and the start of the next menu selection field.

- `horizontal ? (y/n)`
  Enter `y` for a horizontal menu or `n` for a vertical menu.

- `control field onscreen length`
  The onscreen length for the menu control fields.

- `control max (shifting) length`
  The maximum length for the menu control fields.

The two arrays will be separated by one space. Both arrays will contain the number of elements specified in the `Number of entries` field. The first array will be the selection array, and the second will be the control string array. Return to draw mode to enter the selection strings into the first array and the control strings into the second.

Since the shortcut menu building feature creates an array, it should not be used to create a menu that will have submenus. However, the shortcut method can be used to create the submenus themselves. See page 46 for a full description of menus and submenus.

Shortcut Group Creation

Shortcut group creation allows you to create a vertical or horizontal radio button group or checklist. A checklist is a group of fields of which any number may be selected. A radio button group is a group of fields of which exactly one (no more, no less) is always selected. A selection is made by pressing the NL key, or by pressing the first letter of the selection text. Groups are described in greater detail in the Authoring Reference.

Selecting `group` from the Create Special Objects Screen will display the Group Shortcut screen shown below.

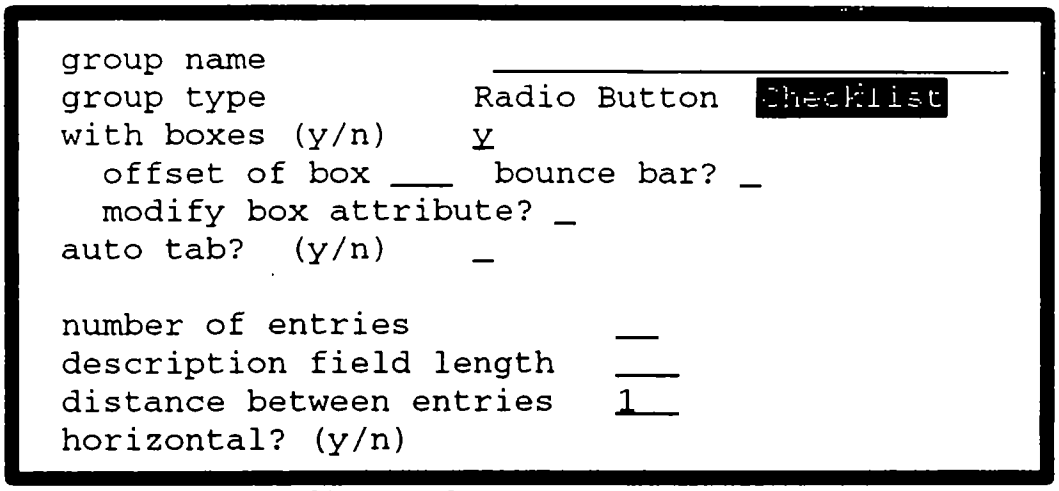


Figure 55: Group Shortcut Screen

Complete the entries in the Group Shortcut Screen as follows:

- `group name`

  Group names follow the same rules as field names; they can be up to 31 characters long, and must start with an alphabetic character. A group and a field may not share the same name on a screen. Note that groups may not be referenced by number.

- `group type`

  Radio Button or Checklist. The group type field is actually a radio button group. Position the cursor with the space bar, then press NL to make the selection.

- `with boxes? (y/n)`

  Enter `y` to create a selection box to the left of each selection field. Selected fields will have an `X` displayed in the box; the selection box character is defined in the video file. You will be prompted for the `offset of box`, `bounce bar?`, and `modify box attribute?` characteristics. See the examples later in this section. Note that, although boxes are displayed on the screen, they are not fields (they do not appear in DRAW mode). If a box happens to overlap a field, then the box will not be displayed.

- `offset of box`

  The number of spaces between the box and the selection text.

- `bounce bar?`

  By default, the cursor is displayed in the box. Type `y` to also display the selection text in toggled reverse video.

- `modify box attribute?`
  Type `y` to display the Display Attributes Screen in order to change the display attributes of the box.

- `auto tab? (y/n)`
  By default, the cursor does not move when a selection is made. Enter `y` to simulate a TAB when a selection is made. The default effect of a TAB is to leave the group. This default can be overridden with the next group attribute.

- `number of entries`
  The number of onscreen group selections. An array with this number of elements will be created.

- `description field length`
  The length of the group selection fields (they will contain the text of the group selections).

- `distance between entries`
  The distance between group entries, measured in columns for horizontal groups and in rows for vertical groups. For horizontal groups, the distance is measured between the end of a group selection field and the start of the next selection box (or group selection field if the group has no boxes).

- `horizontal ? (y/n)`
  Enter `y` for a horizontal group or `n` for a vertical group.

Each example below shows a group with three choices: eggs, noodles, and rice (also known as the selection text). In each example, eggs are selected, and the cursor is on noodles. The examples use reverse video and blinking attributes (blinking in the example is indicated by grayed out text). The actual attributes depend on the capabilities of the video display. Furthermore, the examples apply equally well to radio buttons and checklists.

Figure 56: Group With No Boxes

Figure 57: Group With Boxes and Bounce Bar

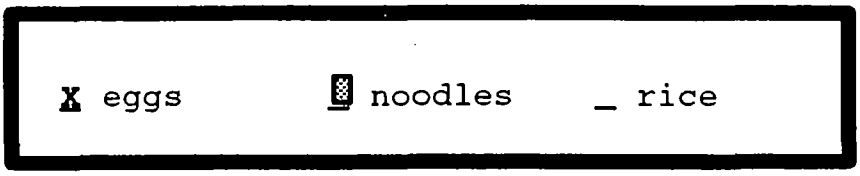


Figure 58: Group With Boxes and No Bounce Bar

Screen Name Field Creation

It is often convenient to have the name of a screen appear somewhere on the screen. If the name were entered as display text, it would have to be changed manually if the screen were renamed or copied. **JAM** provides a method for creating a screen name field, which is simply that any field whose name is `jam_name` will automatically pick up and display the name of the screen from the operating system at runtime. Note that the name will not display in draw or test mode.

Selecting `screen name field` from the Create Special Objects Screen, automatically creates this field and returns the Screen Editor to draw/test mode. By default, the field is protected from entry or clearing. The upper and lower case field edits can be used to control the case in which the screen name is displayed. Note that the file name suffix (if any) specified by the configuration variable `SMFEXTENSION` is not displayed in the screen name field. Therefore, if `SMFEXTENSION` is `.jam`, then a screen stored in a file named `mainmenu.`***jam*** would be display in the `jam_name` field as `mainmenu`.

4.3.11

Field and Group Names (SPF3)

The PF5 and SPF6 keys display the name of an individual field or group. Sometimes, it is convenient to see all field or group names at once. Press SPF3 to see all field names at once. Each field on the screen, even if non–display, is displayed with its field name in place of its normal content. The field name is truncated to the length of the field. An example screen is shown below.

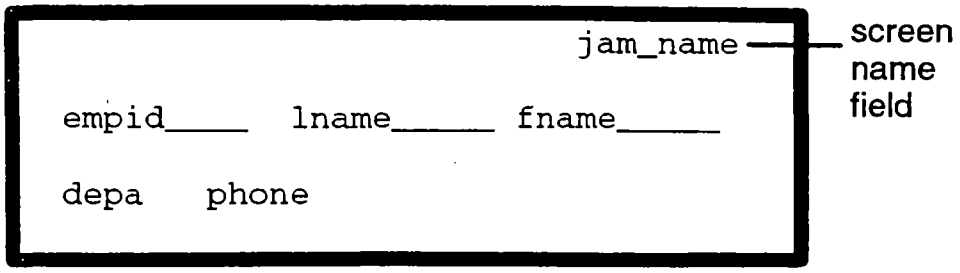


Figure 59: Screen With Field Names Visible

Names longer than the field length are truncated (e.g. depa above). At this point, any function key other than PF2 will return the Screen Editor to draw/test mode. The PF2 key is a toggle between field name and group name display; pressing the PF2 key once will replace the field names with group names. Pressing PF2 again will replace the group names with field names.

4.3.12

Data Dictionary Search (SPF4)

Use the SPF4 key to create a field from a data dictionary (DD) entry, or to check the consistency between a field and its corresponding data dictionary entry. To create a field, press SPF4 while the cursor is not in a field. The Data Dictionary Search Screen is displayed, as shown below.

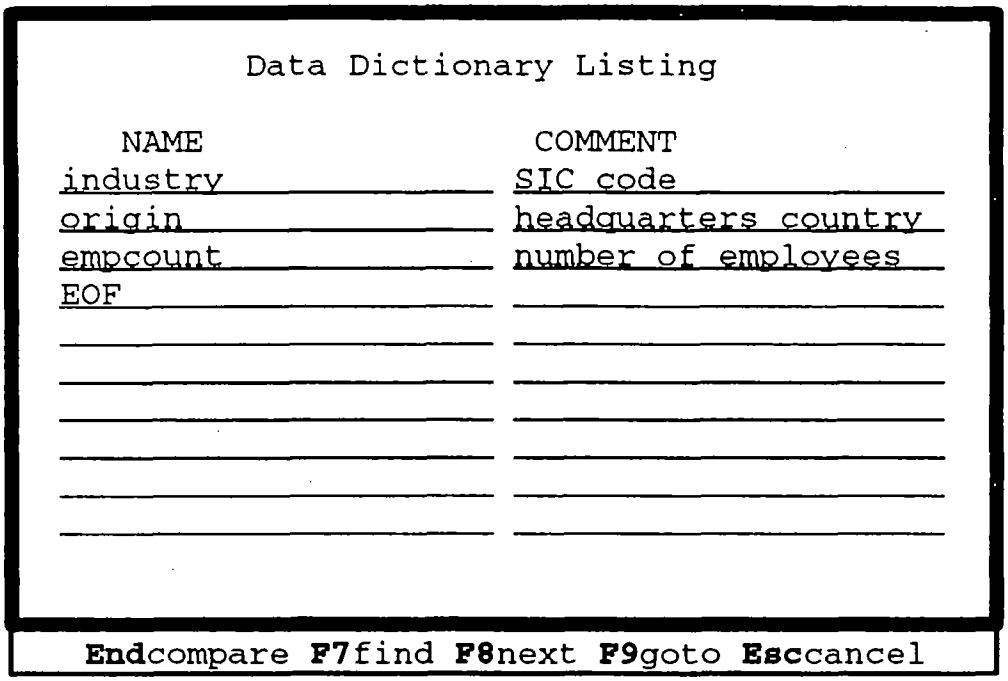


Figure 60: Data Dictionary Search Screen

Select an entry, and press XMIT; a field with the characteristics specified in the data dictionary will be created on the screen at the current cursor position. PAGE DOWN, PAGE UP, PF7 (find), PF8 (next), and PF9 (go to line number) may be used to select the desired entry. The Data Dictionary Editor chapter contains a more detailed description of this screen.

To compare a field with a data dictionary entry, press SPF4 while the cursor is in a field. The Data Dictionary Search Screen is displayed. When an entry is selected, the attributes of the entry and the field will be compared on the Data Dictionary Comparison Screen shown below.

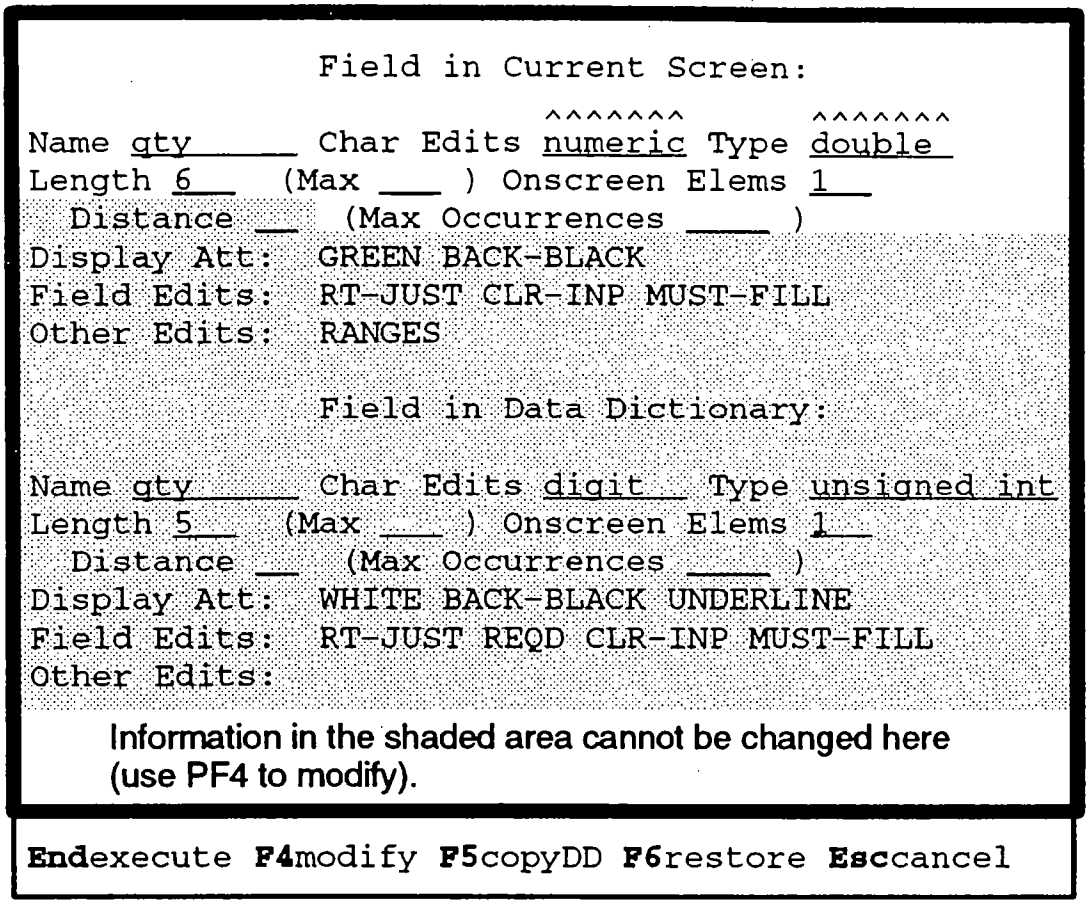


Figure 61: Data Dictionary Comparison Screen (Note: for readability, the screen shown here is not exactly as it appears in **JAM**.)

A field's (not an entry's) characteristics can be changed from within the Data Dictionary Comparison Screen. Press PF4 to change a characteristic that cannot be changed directly from the Comparison screen. Press PF5 to change the characteristic of the field under the cursor to the characteristic defined in the DD. Press PF6 to restore the characteristic of the field to its original state.

Fields that are not in the shaded area can be changed directly. To change the `character edits` or `type`, move the cursor to that field and press the up or down cursor keys (or the newline key) to cycle through the possible choices. To confirm the changes press XMIT; to abort, press EXIT.

4.3.13

Add to Data Dictionary (SPF5)

Use the SPF5 key to create a data dictionary entry from a field. Place the cursor in a field and press SPF5. An entry in the Data Dictionary will be created that has all of the

field's characteristics. An error will result if the cursor is not in a field, if the field is not named, or if the name already exists in the DD.

The entry's scope (see page 100) will be the scope defined as the default for newly created DD entries. The entry will not become part of the local data block in application mode until the `rebuild index` command is selected from within the DD editor (see page 5.3) or until the application is re-started.

4.3.14

Group Attributes (SPF6)

Use SPF6 to create a group from existing fields, to add or delete fields to/from a group, or to change the attributes of a group. Groups were discussed briefly under SPF2 (page 85), where a shortcut technique for group creation was described. Additional information is available in the the Authoring Reference chapter.

A group is a set of fields, one or more of which may be selected by the user. While the value of a field is the data contained by the field, the value of a group is the list of selected fields in the group. There are two types of groups: radio buttons and checklists. A radio button group has exactly one selected field at all times; selecting a field automatically de-selects the previously selected field. A checklist can have any number of its fields selected.

To create a group, first create all the fields that are to be part of the group. You can create the group before creating any fields, but it is usually easier to create the fields first. The fields can be placed close to each other, or far apart. The fields can be part of one or more arrays (possibly scrolling). Each occurrence of each array is individually selectable.

After the fields are created, position the cursor in one of the fields and press SPF6. Alternatively, to add or drop fields from an existing group, or to change the attributes of an existing group, position your cursor in a field which is a member of the group, and press SPF6. The Group Attributes Screen will be displayed as shown below.

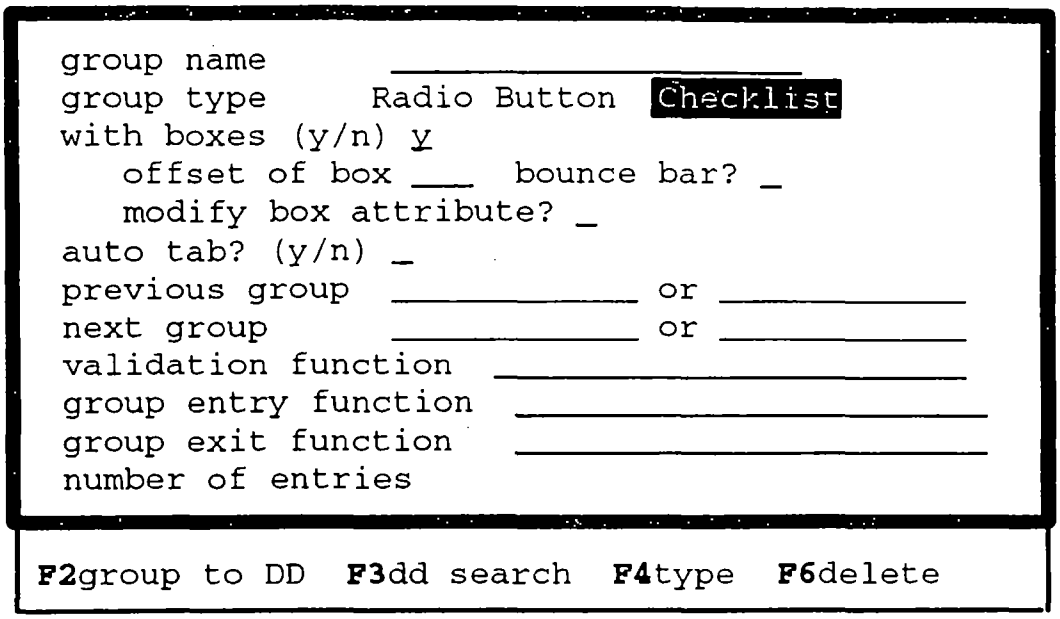


Figure 62: Group Attributes Screen

The entries on the Group Attributes Screen are described below:

- `group name`

  Group names follow the same rules as field names; they can be up to 31 characters long, and must start with an alphabetic character. A group and a field may not share the same name on a screen. Changing the group name will not delete the group; the selected fields will be moved from the existing group to the new group. Note that groups may not be referenced by number.

- `group type`

  Radio Button or Checklist. The group type field is actually a radio button group. Position the cursor with the space bar, then press NL to make the selection.

- `with boxes? (y/n)`

  Enter `y` to create a selection box to the left of each selection field. Selected fields will have an `X` displayed in the box; the selection box character is defined in the video file. You will be prompted for the `offset of box`, `bounce bar?`, and `modify box attribute?` characteristics.

- `offset of box`

  The number of spaces between the box and the selection text.

- `bounce bar?`
  By default, the cursor is displayed in the box. Type `y` to display the cursor by showing the selection text in toggled reverse video.
- `modify box attribute?`
  Type `y` to display an Attributes Screen in order to change the display attributes of the box.
- `auto tab? (y/n)`
  By default, the cursor does not move when a selection is made. In a radio button group, setting the autotab attribute to `y` will cause the cursor to leave the group when a selection is made. In a checklist, this attribute will cause the cursor to move to the next item in the group, just as if the space bar were pressed.
- `previous group`
- `next group`
  The name of a group or field to move to when BACKTAB (for previous group) or TAB (for next group) is pressed. These may be specified in the same manner as the next and previous field designations discussed on page 50.
- `validation function`
- `group entry function`
- `group exit function`
  Hook strings for calling group hook functions. The hook string includes the name, and optionally the arguments, of the hook function. **JAM** guarantees that the group exit function will be called exactly once for each call to the group entry function. The validation function is called when the group is exited with the TAB key, and when XMIT is pressed. A validation function of a field within the group is called only when the field is selected (the same is true of fields that are part of a menu). For checklists, the field validation function is also called when a field is deselected. The *Programmer's Guide* describes the hook functions in greater detail. The syntax of these hook strings is the same as the syntax of field function hook strings, described on page 59.
- `number of entries`
  The number of onscreen group selections. This cannot be changed here. To add or delete entries, follow the instructions given below.

The following function keys are active when the Group Attributes Screen is displayed:

- PF2 Add the group definition to the data dictionary. The group must contain fields in order for it to be added, although the fields themselves are not add-

ed to the DD. In fact, no field information is maintained in the group entry in the DD. The DD entry contains only the group name, number of occurrences and the group's attributes. See page 103 for more information on groups in the data dictionary.

- PF3 Search the data dictionary for a group entry. The Data Dictionary Search Screen will display, but with only groups listed. See page 89 for a description of that screen. On the Data Dictionary Search Screen, press XMIT to copy the attributes of the selected group into the Group Attributes Screen. This can be used to assign attributes to a new group, to compare the attributes of a group with the attributes stored in the DD, or to update the attributes of a group from the DD.
- PF4 Designate a type for the group so that the `f2struct` utility can include it in a programming language structure for the screen. See page 72 for a discussion of type.
- PF5 Delete the group definition entirely. This does not delete the fields in the group; they revert to not belonging to a group.

When you are done entering or modifying group attributes, press XMIT to select the group of field, or EXIT to abort. Pressing XMIT will save the modified attributes, close the Group Attributes Screen, and highlight the fields that currently compose the group; these fields are selected with highlighted brackets. Move the cursor from field to field, selecting and deselecting fields to be included in the group by toggling with the PF6 key. Press XMIT to place the selected fields into the group with the attributes specified by the Group Attributes Screen. To abort the group selection, press EXIT (this does not undo changes made to the group's attributes). Note that no field can be in two groups.

4.3.15
Synchronized Arrays (SPF7)

Scrolling arrays can be synchronized so that they scroll together. This helps manage related information in table–oriented screens. Any set of scrollable arrays on a screen can be synchronized, provided that they have the same number of onscreen elements and the same maximum number of occurrences.

Parallel arrays are automatically synchronized by **JAM**. Automatic synchronization is discussed on page 70. To manually synchronize a set of arrays, press the SPF7 key while the cursor is on one of the arrays. Using the arrow keys, move the cursor to another scrolling array and press PF6 to select it. Select (or de–select) additional arrays by moving the cursor and pressing PF6. Each selected array will be highlighted. If an array that you try to select has a different number of onscreen elements from the other selected arrays, **JAM** will reject the selection and issue an error message. However, **JAM**

is more forgiving about the maximum number of occurrences. When selecting a set of arrays to synchronize, the Screen Editor will adjust the maximum number of occurrences of all selected arrays to the highest maximum number of occurrences of the selected arrays. This change occurs on the screen.

When you have completed your selection, press XMIT to synchronize the arrays or EXIT to abort.

To de–synchronize one or more (including all) arrays in a set of synchronized arrays, press SPF7 while the cursor is on one of the arrays. All of the arrays synchronized with that array (including itself) will be highlighted. To de–synchronize an array, move the cursor on it and press PF6. Press XMIT to complete the de–synchronization, or EXIT to abort. The only way to de–synchronize parallel arrays is to use the isolate option on the size window (page 71).

4.3.16

Character Graphics (SPF8)

Press SPF8 to add graphics characters to the screen. At that point, the Graphics Selection Screen, filled with character graphics defined for your terminal[15], will be displayed. See the discussion of video files in the *Configuration Guide* for a discussion of how **JAM** determines the content of the Graphics Selection Screen. To select a graphics character and place it on the underlying screen, position the cursor over the graphics character and press XMIT. To insert the character repeatedly, press the PF9 (repeat) key instead of bringing up the window again. Note that character graphics are just characters, and follow the same rules as text characters for display attributes. To draw lines with graphic characters, use the line drawing mode described in the next section.

An example Graphics Selection Screen is shown below.

15. Note that if you design a screen with character graphics on one display and use it on another with a different character graphics set, the screen may look different. This is because when the screen is saved, only the ASCII code for the character in question is stored, and character graphic ASCII codes vary from display terminal to display terminal.

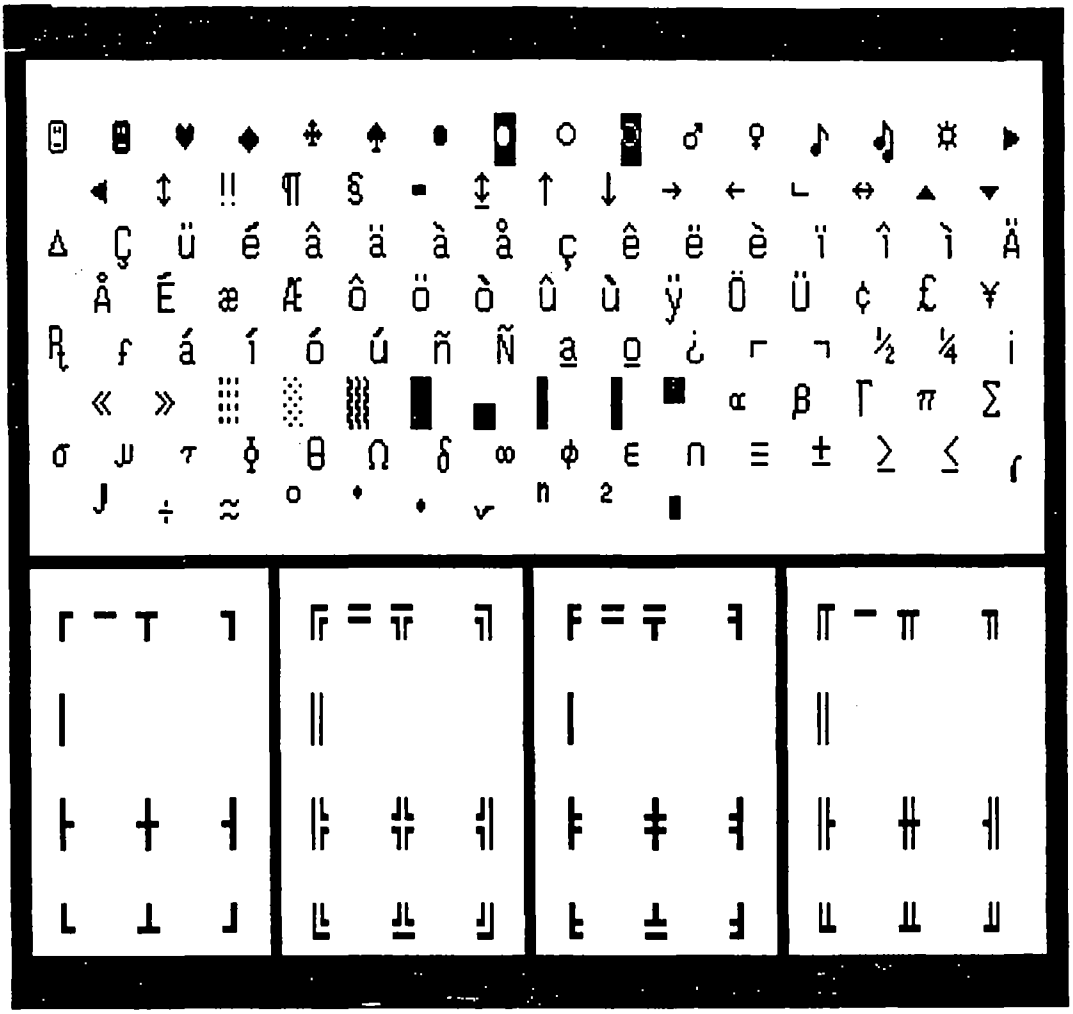
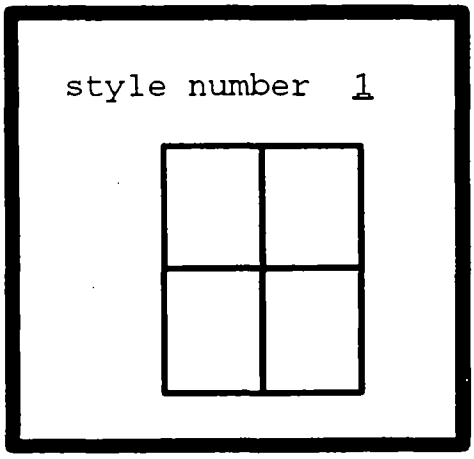


Figure 63: Graphics Selection Screen for IBM PC

4.3.17

Line Drawing (SPF9)

Press SPF9 to draw lines and boxes. The Line Drawing Style Screen is displayed as below. There are ten line graphic styles available, labelled 0–9. When you enter a new style number, the sample box in the screen is redrawn to show what that style looks like.



The appearance of the styles is terminal dependent.

Figure 64: Line Drawing Style Screen

After choosing a style and pressing XMIT, the Screen Editor will be in line draw mode, and the status line will look like:

```
@F2Pen down F3box F4display F5style
```

Figure 65: Line Drawing Mode Status Line

Move the cursor via the cursor keys to draw a line. When the cursor changes direction, a corner character is inserted. When a line crosses another line of the same style, an intersect character appears.

Four function keys are available in line drawing mode: PF2, PF3, PF4, and PF5. PF2 toggles between pen up and pen down. Pen up mode allows you to reposition the cursor without drawing lines. PF3 places toggles into box draw mode, putting an asterisk at the current cursor position to mark one corner of a box. As the cursor moves, a second asterisk moves to indicate the diagonally opposite corner of the box. The box is drawn when PF3 is pressed again. PF4 lets you change the pen's display attributes on the Display Attributes Screen (see page 24). PF5 allows you to change the line drawing style on the Line Graphics Style window.

Please note that line graphics mode is merely a convenient way of placing character graphics on the screen. The screen retains only the ASCII character codes for the lines.

Chapter 5
The Data Dictionary Editor

5.1
INTRODUCTION

In this chapter we present a description of operation of the Data Dictionary Editor, presenting the features in the order that they are encountered by a developer. A data dictionary is a repository of characteristics of fields, groups, and records (records are collections of fields and groups used by several **JAM** library functions). Fields and groups can be copied to and from screens (some restrictions apply to groups). Field and group characteristics can be compared to data dictionary entry characteristics from within the Screen Editor; the `jamcheck` utility compares, and optionally fixes, consistency between all screens and the data dictionary. In addition, **JAM** creates the local data block (LDB) from the data dictionary (DD) at runtime. The LDB holds data that is transferred between screens at runtime. Please see the **JAM** *Overview* for a more complete discussion of the DD and the LDB. Please see the Authoring Reference chapter (page 135) for a discussion of the interaction between the LDB and screens.

A developer generally uses the Data Dictionary Editor in the following ways:

- create new data dictionary entries.
- modify or delete existing data dictionary entries.

JAM looks for the data dictionary in a binary file named `data.dic`. The name that **JAM** looks for can be changed with the environment variable `SMDICNAME` (see the *Configuration Guide*), or with the library function `sm_dicname` (see the *Programmer's Guide*). The data dictionary file may also be edited by converting the binary file to an ASCII file with the `dd2asc` utility, editing it, and converting it back to a binary file with `dd2asc` (see the *Utilities Guide*).

5.2 ENTERING THE DATA DICTIONARY EDITOR (SPF6)

The Data Dictionary Editor can be entered from application mode by pressing SPF6. The display will clear, and the Data Dictionary Maintenance Screen will be displayed as shown below.

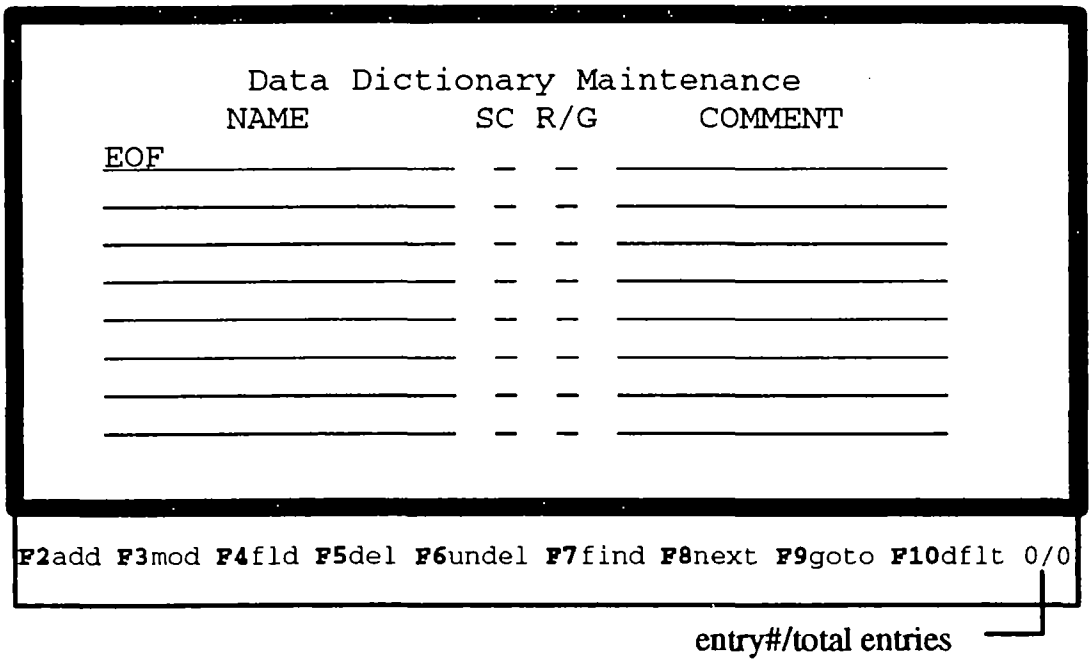


Figure 66: Data Dictionary Maintenance Screen (Note: for readability, the screen shown here is not exactly as it appears in **JAM**.)

If you are creating a new data dictionary file, then the message

ERROR: Cannot read data.dic.

will be displayed. Press the space bar to clear the message. The message will be replaced with a status line showing function keys, as shown above, and you may continue with an empty data dictionary. The Data Dictionary Maintenance Screen has four arrays:

- NAME

  The name of the entry must be unique within the dictionary. It can be any name that is a legal field name (see page 49).

- SC

  The scope of the entry. A scope of 0 prevents the entry from creating a cor-

responding LDB entry. A scope of 1 to 9 groups this entry with other entries for clearing and initializing entries as a group. See page 104 for additional information.

- R/G

  The entry type. Leave blank for a field entry. Type R for a record, or G for a group.

- COMMENT

  Any enterable text. The COMMENT field can be searched from within the DD editor to assist with finding entries.

The cursor is positioned to the left of the current entry. The ARROW, TAB, BACK-TAB, NL, PAGE UP, and PAGE DOWN keys will change the current entry. The function keys shown on the status line perform actions that are described later.

5.3
EXITING THE DATA DICTIONARY EDITOR

To exit the editor, type EXIT whenever the Data Dictionary Maintenance Screen is active. The Data Dictionary Exit Screen is displayed as shown below.

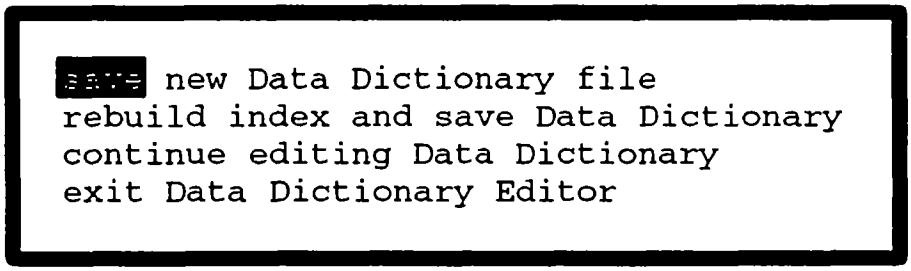


Figure 67: Data Dictionary Exit Screen

The menu in the Data Dictionary Editor Exit Window has the following options:

- save

  Save the data dictionary on disk.

- rebuild

  Save the data dictionary on disk, then re-initialize the local data block. A rebuild is necessary only to continue the authoring session with an LDB that reflects that latest changes to the dictionary. All data in the LDB is lost, and all LDB initialization files (see the Authoring Reference chapter) are

re–read, just as if the authoring utility were re–started from the operating system.

- `continue`
  Return to the Data Dictionary Editor.
- `exit`
  Return to application mode. The data dictionary is not saved — use the `save` option to save the dictionary prior to exiting.

5.4
DATA DICTIONARY EDITOR FUNCTIONS

The functions of the Data Dictionary Editor are accessed via the function keys PF2 through PF10. These keys are displayed on the status line, as shown on page 100. The ordering of this section follows the order of the function keys, as listed below.

- PF2 Add new entries.
- PF3 Modify entries.
- PF4 Modify field, group, or record characteristics.
- PF5 Delete current entry.
- PF6 Undelete most recently deleted entry.
- PF7 Find entry by name or comment.
- PF8 Find next entry.
- PF9 Go to an entry by line number.
- PF10 Set default field characteristics.

5.4.1
Add Data Dictionary Entries (PF2)

To add an entry above an existing entry, position the cursor on the existing entry and press PF2. One or more lines will open up for the addition of new entries, and the cursor will be positioned in the NAME column on the first new line. The current line (the line in which the entry will be entered) will be marked by an asterisk (*). Cursor movement will be restricted to that line. The status line will change to reflect the fact that the function keys will perform different actions. For an empty dictionary, the screen will appear as follows:

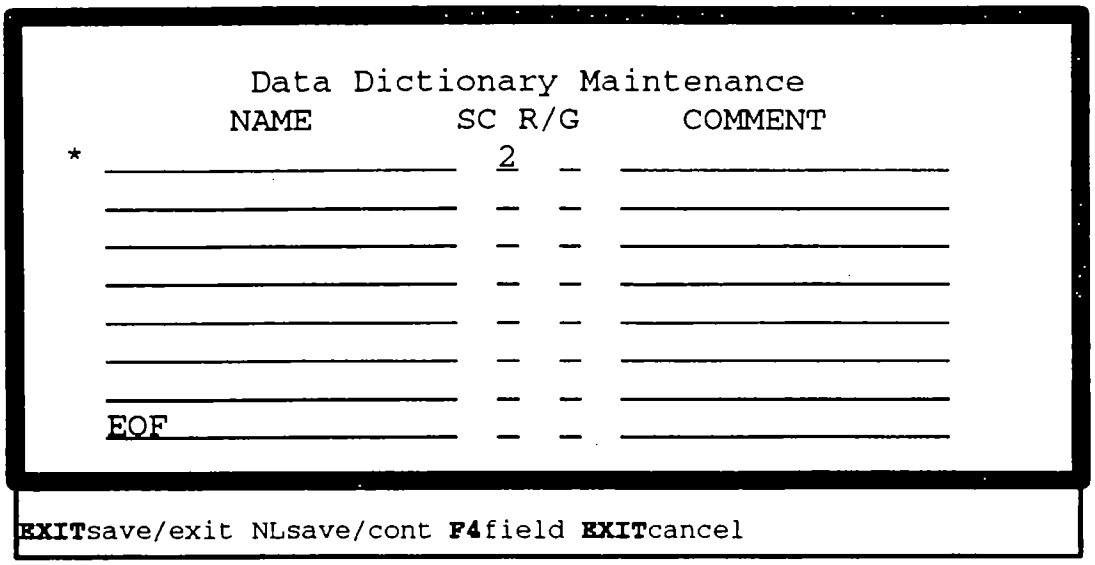


Figure 68: Adding New Data Dictionary Entries (Note: for readability, the screen shown here is not exactly as it appears in **JAM**.)

Enter the entry name, the scope (which defaults to 2), and R or G if the entry is a record or group respectively. Entering R will cause the Record Definition Screen to pop up; records are dicussed below. Entering G will cause the Group Attributes Screen to pop up; see page 93. A comment can be entered if desired. Press XMIT to add the entry to the data dictionary and return to the normal status line, NL to add the entry and continue adding more on the lines below, or PF4 to assign field or group or record characteristics. PF4 displays the Field Characteristics Menu (page 36), Record Definition Screen, or Group Attributes Screen depending on the entry type. To abort the addition of the new entry on which the cursor is positioned, press EXIT.

Data Dictionary Groups

The data dictionary maintains the name and number of occurrences associated with a group, as well as the previous, next, and group functions. It also maintains the group type (checklist or radio button) and box information. There is no information about the fields that belong to the group. These facts have several implications:

- Creating a group from the DD does not create the fields in the group. It is the developer's responsibility to ensure that the group contains the proper fields.
- A group must have at least one field before being added to the DD via the SPF5 (Add to DD) feature of the Screen Editor (page 91). This tells the DD how many occurrences to specify for the group.
- A group entry in the LDB maintains a list of selected fields (identified by occurrence number within the group), not the contents of those

fields. Therefore, a radio button group entry has one occurrence (since only one field can be selected) and a checklist group entry has as many occurrences as there are occurrences in the group.

Scope of Field and Group Entries

Every field and group DD and LDB entry has a scope, although scope is not a characteristic of a field or group on a screen. A DD entry can have a scope between 0 and 9. A scope of 1 through 9 tells **JAM** to create a corresponding entry in the LDB at runtime. A scope of 0 tells **JAM** not to create a corresponding entry at runtime. Therefore, a DD entry with a scope of 0 is used only during development in order to facilitate consistency between fields and groups on different screens.

LDB entries can be cleared and reset as a group, based on scope, via the **JAM** library functions `sm_lclear` and `sm_lreset`. An LDB entry with a scope of 1 is considered to be constant. Entries with a scope of 1 cannot be changed after the LDB is first initialized (see page 109), except via `sm_lclear` and `sm_lreset`.

LDB scopes may, for example, be used to manage the contents of collections of LDB entries. Consider an insurance application that tracks group policies and individuals within each group. The insurance company name and address could be given a scope of 1, since they would not change. Entries relating to a group policy, such as the group policy number, could be given a scope of 2. Entries relating to an individual policyholder, such as the name of the policyholder, could be given a scope of 3. Each time the application begins dealing with a new policyholder, scope 3 entries could be cleared. Each time the application begins dealing with a new group policy, scope 2 entries could be cleared.

Data Dictionary Records

A record is a list of fields and groups. Several **JAM** library functions access the local data block via structures created from data dictionary records. See the *Progammer's Guide* (e.g. `sm_rrecord`) and the *Utilities Guide* (`dd2struct`) for additional information. Note that records are not the primary method of accessing LDB entries programmatically; the primary method is to access them like fields via `sm_getfield` and `sm_putfield`.

Records are created and modified on the Record Definition Screen shown below. The screen is displayed when R is entered into the type field, or when PF4 is pressed while the cursor is on an entry that defines a record.

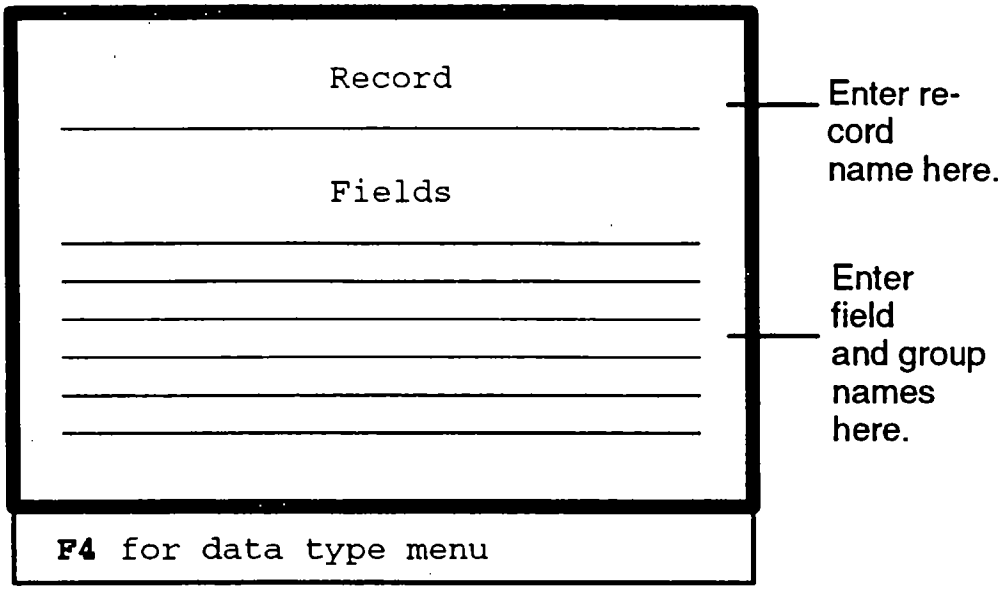


Figure 69: Record Definition Screen

To create a new record, enter the record's name and the names of the fields and groups that compose the record. Note that a record should not be a component of a record. Although the Data Dictionary Editor allows this, the `dd2struct` routine will create an error. The rules for constructing a field name (see page 49) apply to constructing a record name. The name of an existing record cannot be modified on this screen; use PF3 instead.

You may optionally specify a programming language data type for any given component in the record. The language data type impacts the programming language structure generated by `dd2struct`. While the cursor is placed on the component name, press PF4 to pop up the Data Type Menu (see page 73). If you do not specify programming language data type for a record component, the data type of the named data dictionary entry will be used.

Press XMIT to save the record components or press EXIT to abort.

5.4.2

Modify Existing Entries (PF3)

To modify any characteristic of an existing entry, position the cursor to that entry and press PF3. You may modify any characteristic, including the name, scope, type, and comment. This differs from pressing PF4 in that name, scope, type, and comment cannot be modifed with PF4. All function keys work exactly as when adding new entries.

5.4.3

Modify Field Characteristics (PF4)

PF4 is a shortcut for pressing PF3 to modify an entry, followed by pressing PF4 to change that entry's field, group, or record characteristics. Striking the PF4 key from the top level of the Data Dictionary Editor is a shortcut for striking PF3 to modify the current entry and then PF4 to modify field characteristics.

5.4.4

Deleting and Undeleting Entries (PF5, PF6)

To delete an entry, position the cursor next to it and press PF5. The entry will be immediately deleted without asking for confirmation. The most recently deleted entry can be restored with the PF6 key. When you press PF6, a line opens above the cursor and the most recently deleted entry is inserted there. You can use PF5 and PF6 to move items around in the data dictionary, but be careful. PF6 will undelete only the last deletion. If you delete two entries in a row the first of the two will be unrecoverable (except, possibly, by aborting the editing session).

5.4.5

Searching for Entries (PF7, PF8)

Press PF7 to search for a data dictionary entry. The Search String Screen will be displayed as shown below.

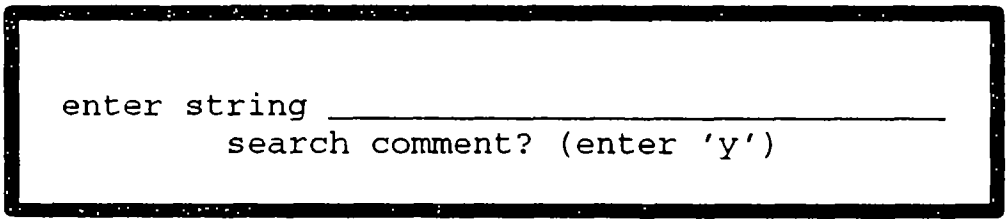


Figure 70: Search String Screen

To search for a string, simply type in the text. The search string, in addition to ordinary text, may contain certain special characters:

- To search for a name that begins with a sequence of characters, enter those characters preceded by a caret (^).
- The question mark (?) is a wild card, matching any single character.

- The asterisk (*) is a wild card, matching any string of zero or more characters.

These wild card characters do not have the same meaning when used in regular expressions for field validation. Here are some examples of search strings:

- The search string `^abc` finds the first entry after the cursor position that *begins* with `abc`. If the caret were missing from the search string, the first entry that contained the string `abc` would be found.
- To find an entry whose name contains `xyz` but whose beginning characters are unknown, enter `xyz` as a search string. This will also find strings with `xyz` in the beginning.
- To find an entry whose first character is `a`, whose second character is `b` or `c`, whose third character is `d`, and has a `z` somewhere following the `d`, enter the search string `^a?d*z`. This string would also match entries whose second character is other than `b` or `c`.

To search the comments rather than the names of entries, enter `y` in the `search comment? (enter 'y')` field.

A search may be repeated by striking the PF8 key. Searches start at the current cursor position and go to the end, then wrap around to the beginning of the data dictionary and continue to the line above the current position.

5.4.6
Go to a Specified Line (PF9)

To go to an entry specified by line number, press the PF9 key. You will prompted for the desired number in the Line Number Screen shown below.

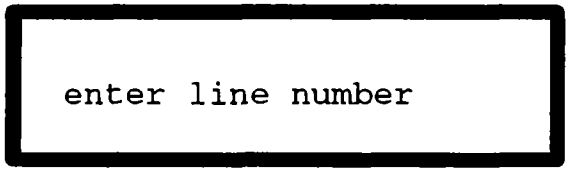


Figure 71: Line Number Screen

Entering 0 or 1 will position the cursor on the first line of the data dictionary. Entering a number greater than the total number of entries will position the cursor at the end of the data dictionary (i.e. at the EOF marker).

5.4.7

Default Entry Settings (PF10)

When an entry is added to the data dictionary, it is given default characteristics. To change the default characteristics, press PF10 to display the Data Dictionary Defaults Screen shown below.

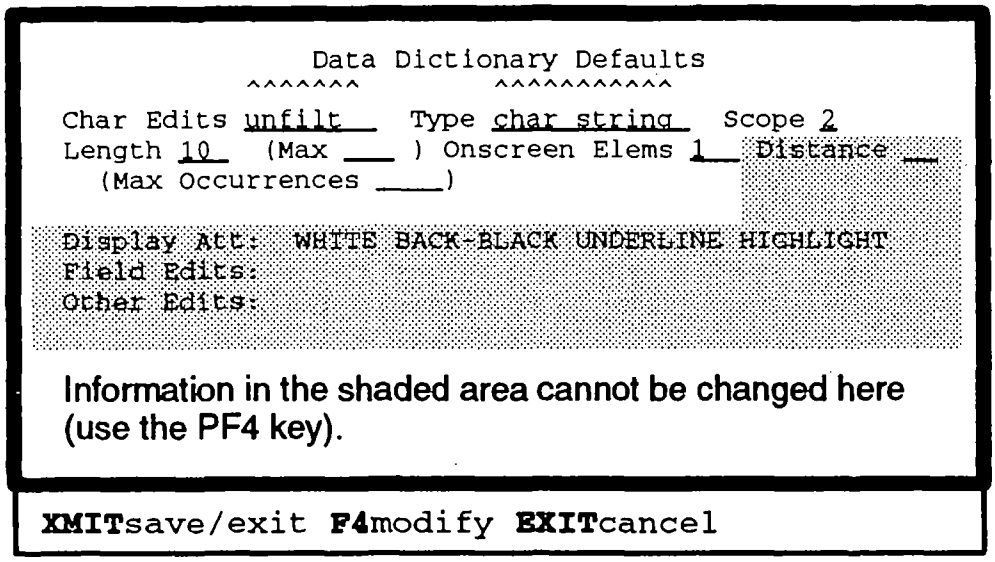


Figure 72: Data Dictionary Defaults Screen (Note: for readability, the screen shown here is not exactly as it appears in **JAM**.)

The following characteristics can be set:

- `Char Edits`
  This is a circularly scrolling array that defines the character edit. Use up and down arrow keys to display the desired edit and then tab out. Whatever is left in the field when XMIT is pressed is selected.
- `Type`
  This is a circularly scrolling array that defines the data type.
- `Scope`
  The scope of the data dictionary entry. Scope is a digit between 0 and 9.
- `Length`
  The onscreen length.
- `(Max )`
  The maximum length. If the field is copied to a screen, and the maximum length is greater than the onscreen length, then the field will shift.
- `Onscreen Elems`
  The number of onscreen array elements.

- `Distance`
  The number of lines or columns between elements that are vertical or horizontal respectively.

- `(Max Occurrences)`
  The maximum number of occurrences.

To set defaults for other characteristics, press the PF4 key while the Data Dictionary Defaults Screen is displayed. The Field Characteristics Menu (see page 36) will pop up. When you are done with setting defaults, press XMIT to save the new settings or press EXIT to discard them.

5.5
LDB INITIALIZATION

As discussed on page 104, the LDB is created from DD entries having a scope between 1 and 9. By default, all LDB entries are empty. The LDB is created and initialized by a call to `sm_ldb_init`, which generally occurs before `sm_jtop` is called in the main routine provided with **JAM** (see the *Programmer's Guide*). Entries may be initialized to specified values via LDB initialization files.

An initialization file is a text file that contains a list of pairs of an LDB entry name and a value. Each pair must be on a separate line. Each LDB entry name and value must be surrounded by quotation marks. For example, to initialize the LDB entry company_name with the value JYACC, the initialization file should contain the following line:

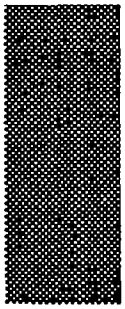
```
"company_name" "JYACC"
```

Individual occurrences of entries can be initialized by subscripting the entry name as shown below.

```
"co_city[2]"    "Paramus"
```

JAM searches the directories specified in `SMPATH` for the LDB initialization files. Any of the files can initialize variables of any scope, but by convention, `const.ini` and `global.ini` initialize entries of scope 1, `tran.ini` initializes entries of scope 2, and `local.ini` initializes entries of scope 3. These file names can be changed using the `SMININAMES` configuration variable. See the *Configuration Guide* for details. A complete example initialization file is shown below:

```
"company_name" "JYACC"
"co_street[1]" "116 John Street"
"co_street[2]" "12 Route 17 North"
"co_street[3]" "55 William Street #200"
"co_city[1]"   "New York"
"co_city[2]"   "Paramus"
"co_city[3]"   "Wellesley"
"co_state[1]"  "NY"
"co_state[2]"  "NJ"
"co_state[3]"  "MA"
"co_zipcode[1]"    "10038"
"co_zipcode[2]"    "07652"
"co_zipcode[3]"    "02181"
```



Chapter 6
The Keyset Editor

6.1
INTRODUCTION TO SOFT KEYS

Soft keys are physical keys that may have more than one logical value. Labels on the terminal display the current logical value of each key. You may define several rows of soft keys, but only one row will be displayed at a time. The logical translation of a soft key varies, depending on the row of labels which appears on the monitor when the key is pressed. Often, one soft key is given the function of changing rows. Thus the end–user can change the value of a key. In this sense soft keys are different from regular or "hard" keys.

Soft keys can be particularly useful when a terminal has few function keys. They allow the keys to be "multiplexed," with the same key serving multiple functions, even on the same screen. Another advantage of soft keys is that they are "self–documenting," so as the user moves between rows, or from screen to screen, the labels change to indicate the function of a key in a given context.

The diagram in Figure 73 below, shows a sample screen with soft keys.

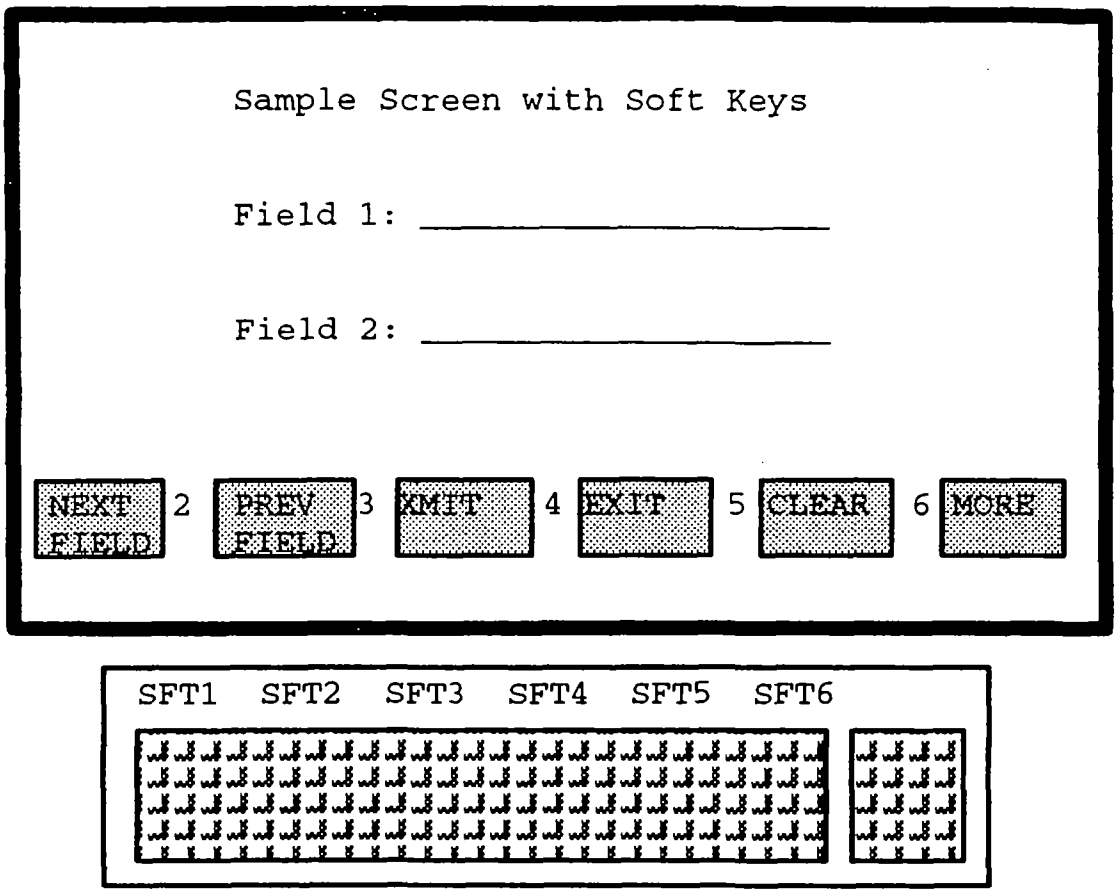


Figure 73: The screen labels indicate the function of soft keys.

Some terminals, particularly HP and AT&T models, provide hardware support for soft keys. On terminals with many function keys, the soft key labels may be used for help text. If your terminal does not provide hardware support for soft keys, **JAM** can simulate the functionality of soft keys, although the area available for screens will be slightly reduced.

6.2
KEYSETS

A *keyset* supplies the mapping of soft keys into logical keys. It also contains the text that displays in the label areas on the screen. There may be more than one row of soft keys in a keyset. Each row has a particular mapping and set of labels. Keysets are created using the *keyset editor.*

You may have more than one keyset in an application, for example an application–wide keyset and several screen specific keysets. Screen–specific keysets are designated in

the `keyset` field of the Screen Editor's Screen Characteristics Screen, as explained on page 35 of the Screen Editor chapter. The translation of soft keys changes depending on which keyset is currently in use. In the absence of a keyset, soft keys have default translations and labels.

6.2.1

The Keyset Editor

The keyset editor is entered from application mode of `jxform` by pressing SPF4. You may also enter the editor by pressing SFT4 (soft key 4)[16]. The entry and exit screens are similar to those of the Screen Editor.

The main screen for the keyset editor is shown in Figure 74 below.

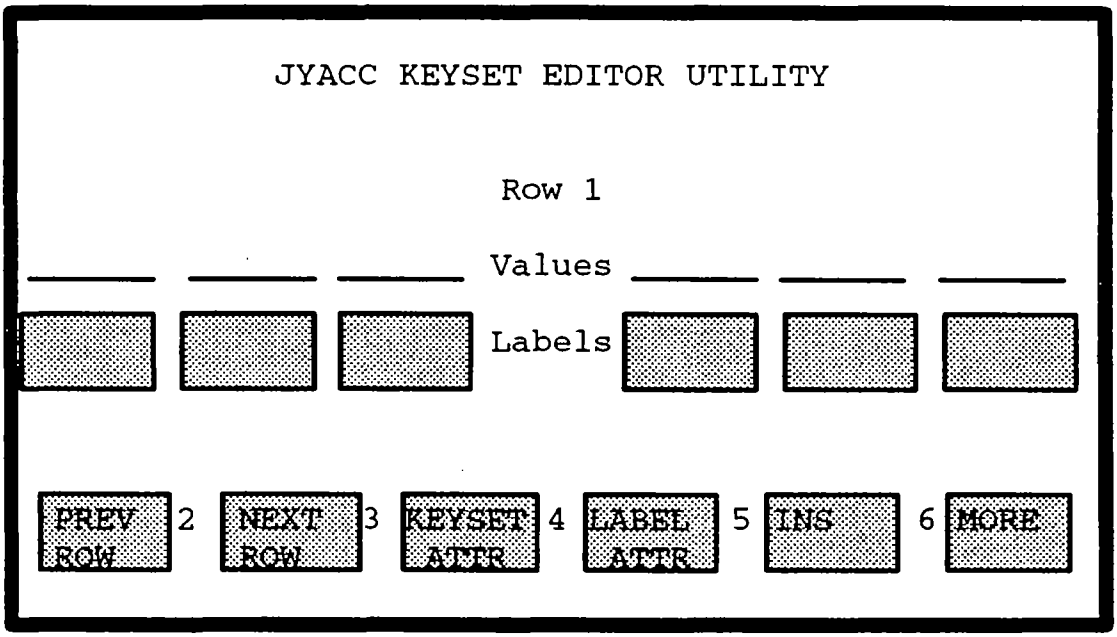


Figure 74: The Keyset Editor screen. Enter soft key values and labels here.

Notice that the diagram in Figure 74 shows six numbered labels across the bottom of the screen. This is the keyset editor's base keyset. The number of labels you will see depends on your *video file*, discussed on page 117.

The editor displays one row of programmable soft keys at a time. There may be up to 24 soft keys in each row. Use the blank lines in the center of the screen to enter the soft

16. The Keyset Editor is available from the authoring environment only when it is linked in to the authoring executable. **JAM** is generally shipped without the keyset editor linked in, as only those using terminals that support soft keys will generally have need of it. If the keyset editor is not linked in to your authoring executable, you will need to modify `jxmain.c` and re–link `jxform`. See the *Programmer's Guide* for details.

key values, and the blank label areas to enter the labels. You may scroll from one row to the next with the PAGE UP and PAGE DOWN keys.

Entering A Soft Key Value

The value is what **JAM** translates the key press into. A soft key value may be any of the following:

- The logical name of any **JAM** key. It may be entered in upper or lower case; if recognized it will be changed to upper case. The acceptable logical key names are:

EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL
BACK	HOME	DELE	INS	LP	FERA	CLR
SPGU	SPGD	LARR	RARR	DARR	UARR	REFR
EMOH	INSL	DELL	ZOOM	MTGL	LSHF	RSHF
SFTS	VWPT	PF1–PF24	SPF1–SPF24		APP1–APP24	

- The logical name of one of the special soft key navigational functions:

  1. SFTN (go to the Next row of soft keys)
  2. SFTP (go to the Previous row of soft keys)
  3. SFT*x* (go to row *x*. *x* is a value between 1 and 24)

  These special functions can only be assigned to soft keys; they cannot be defined in the keyboard translation table (page 116). SFTN is sometimes referred to as the MORE key. Pressing the MORE key brings up the next row in the keyset. On the last row of soft key labels, the MORE key brings up the first row, in a circular fashion. SFTP performs the opposite function, bringing up the previous row of soft keys. SFT*x* brings up row *x* of the keyset. *x* may be a number between 1 and 24. Note that there is no limit to the number of rows in a keyset, only to the values of the navigational function.

- Any number (entered in decimal, hex or octal format). If you want the soft key translated to a graphics or international character, enter the ISO number here.
- Any single character (not in quotes).

An entry which does not satisfy one of these 4 categories will be treated as an error, and a message will be displayed.

Entering A Soft Key Label

A soft key label consists of 2 lines of up to 8 characters each. It contains the text that will be displayed on the screen when this keyset is in operation. Usually the label text

contains the key's name and/or a description of the key's function. If possible, make the first line of each label unique among the keyset, as the user may have only 1 line available for labels.

The length of the labels is specified via the PF3 function key (see below). On terminals where the video file specifies fewer characters than the keyset editor does, the labels will be truncated (see page 117 for more on the video file).

6.2.2

Keyset Editor Function Keys

The following function keys are in operation in the keyset editor:

- PF3 — displays a keyset global configuration window which allows the developer to change:
  1. the number of soft keys in each row of the keyset.
  2. the length of the labels.
  3. the default display attributes of blank labels.

  Be careful when changing the global configuration of a partially or fully constructed keyset, as it is possible to lose data (for example by shortening the length of the labels). If you make a modification in this window, **JAM** will show you a "preview" of your changes when you exit the window. The resized keyset is displayed read–only, and you may scroll through the various rows with the PAGE UP and PAGE DOWN keys. This preview feature can be very useful for viewing how a keyset might appear on a different terminal. Press TRANSMIT to accept the changes, or EXIT to cancel them.

- PF4 — displays an attribute window for changing the attributes of the label at the current cursor position. All attributes (and colors) are listed even though the terminal may not support them. This permits creation of a keyset on one terminal that will be used on another. We suggest setting the reverse video attribute, so the soft keys labels will appear inside of a rectangle on any terminal. Note that PF4 cannot be used to alter blank labels; use PF3 for that.

- PF5 — inserts a blank row in front of the current row.

- PF6 — deletes the current row. Use this key with care, as there is no "undo" feature. (You can always exit without saving, however.)

- PF7 — prompts for a row number and then moves (not copies) that row in front of the current row.

- PF8 prompts for a row number and then copies that row in front of the current row. This feature can be used to propagate a template row to new rows where, for example, each row has MORE on the left and EXIT and HELP on the right.
- PF9 repeats a "change attribute" (PF4) operation. If the attributes of a soft key are changed, the PF9 key, when pressed on another soft key, will cause its attributes to be similarly modified.
- INSL (INSert Line) inserts a blank soft key description at the current cursor location, shifting subsequent descriptions one position to the right.
- DELL (DELete Line) deletes a single soft key description at the current cursor location, shifting subsequent descriptions one position to the left.

In addition, NL, TAB, BACKTAB and the cursor keys can be used to move around the editor screen. Scroll PAGE UP moves to the previous row of keys, and PAGE DOWN moves to the next row in a circular fashion. Scrolling will generate one (and only one) blank row of keys for input.

6.3
KEYBOARD TRANSLATION TABLE

The keyboard translation table (keyboard mapping) used by **JAM** is terminal specific. It is constructed using the `modkey` utility. The utility `key2bin` is then used to generate the binary file that is actually used by **JAM**.

If a terminal supports soft keys, they should be defined using `modkey`. `modkey` allows up to 24 soft keys. Their logical names are SFT1 through SFT24. Most terminals which support soft keys use 8 keys, although some emulators support 12.

If a terminal does not provide hardware support for soft keys, you may still want to define specific keystroke sequences (for example Alt–1 through Alt–8) as your soft keys. If you do not define them, then soft keys will not be available.

NOTE: Be sure your keyboard translation table has a SFTS (Soft key set) function key defined. SFTS toggles between the user–defined keyset and the system keyset when in application mode of `jxform`. If you do not have a SFTS key defined, you will not be able to test your keyset.

6.4
SELECTION OF KEYSETS

As mentioned above there is always a default keyset. The default keyset has one row. The soft keys SFT1 through SFT24 are translated to PF1 to PF24, and the labels read: `f1` through `f24`.

Most applications will load an *application–level* keyset as part of their initialization (in `jmain.c`). Each screen may also specify a *screen–level* keyset that overrides the application–level keyset for the duration of the screen. Screen–level keysets are stacked as windows are opened. If an open screen has a screen–level keyset and a window without one is opened, then the screen–level keyset (not the application–level keyset) is in use. At any time, an application program may override the keyset currently in use.

Keysets are referenced by name as disk files, memory resident "files", or members of a library. The search rules used by **JAM** to find a keyset are identical to those used to find a screen. A keyset may be made memory resident by using the `bin2c` utility and then registered to **JAM** by means of the form list. Keysets may be placed in libraries, either together with screens and JPL procedures or in separate keyset libraries.

Two additional keyset scopes are the *system–level* keyset, used by `jxform`, and the *override–level* keyset which is displayed only by **JAM**. **JAM** uses the override–level keyset during error messages, query_msg, etc.

Because a keyset can consist of several rows, the keyset is tailored to a specific class of terminals (those with the given number of soft keys). As explained below, keysets may be used on terminals with a different number of soft keys. Generally this presents no problems; however in some rare cases it may be necessary to have different keysets for different terminals. This can easily be controlled by putting keysets in libraries and specifying which library is desired in the setup file. Alternatively, different keysets may be placed in different directories and then the `SMPATH` configuration variable can be used to control which keyset is selected.

6.5
VIDEO FILE

A simple alteration to the video file must be made in order to display the soft key labels. In the absence of a video file entry, no labels will be displayed but the soft keys will still be processed.

6.5.1

The KPAR Statement

The video file entry KPAR (Keyset PARameters) enables soft key labels. An example is shown below:

```
KPAR= LENGTH=8 NUMBER=8 SIMULATE ATTRIBUTE 1LINE
```

The keywords LENGTH, NUMBER and ATTRIBUTE apply to both simulated and hardware supported soft keys:

- LENGTH indicates the length of each label;
- NUMBER indicates how many labels can display in one row on the terminal;
- ATTRIBUTE indicates that the soft key labels can utilize the attribute settings of the terminal.

SIMULATE and 1LINE apply only to simulated soft keys:

- SIMULATE designates that the soft keys are simulated.
- 1LINE tells the soft key simulator to reserve only one line for soft keys. (The default is to reserve 2 lines.)

Be careful not to specify more function keys in your video file than will fit on the screen. If you do, then the last few labels on the right will not appear on the display, but the soft keys associated with them will still be active. A good rule of thumb is that the number of columns on your terminal must be greater than: NUMBER x (LENGTH + 1). Therefore 8 keys of length 8 will fit on an 80 column monitor, but 9 keys of length 8 will not.

6.5.2

The KSET Statement

If the terminal you are using provides hardware support for soft keys (i.e. — they are not simulated), then you also need to specify the sequence that the soft keys should send to the terminal. This video file entry is called KSET. It is passed up to 15 parameters. The first is the soft key number, from 1 to NUMBER. The second and third are pointers to the two label lines. They point to null–terminated strings of length LENGTH. The next 12 parameters are the attribute parameters in precisely the same format as passed to SGR and ASGR. See the *Configuration Guide* for details. The following sequence for the HP2392A terminal is typical:

```
KSET= ESC & f %d k 16 d 0 L %s %s ESC & j B
```

6.6
SIMULATED SOFT KEYS

Soft keys may be simulated on terminals that do not provide hardware support. Simply reserve one or two lines on the bottom of each screen in your application and set up the video file using the `SIMULATE` keyword in the `KPAR` statement as described above, in section 6.5 You do not require a `KSET` statement in your video file if you are simulating soft keys.

6.7
KEYSET PORTABILITY CONSIDERATIONS

A keyset specifies a number of soft key labels in each row. The video file specifies the number of soft keys that are physically available on the terminal in use. The following rules are used to map a keyset to a terminal when the number of soft keys they specify does not match.

If the terminal (as specified in the video file) has more soft keys than are specified in the keyset, each row on the terminal will contain several complete rows from the keyset. For example, if a terminal has 10 soft keys and is displaying a keyset with 4 soft keys per row, the display will show 8 soft keys (2 rows) at a time.

Keep in mind that calls to the SFTN function will bring up the next row of displayable labels. Calls to SFTx will bring up row 'x'. Row 'x' is defined in terms of the display, not in terms of the keyset editor's specifications. Therefore, when the terminal has more soft keys than specified in the keyset editor, SFTx may not perform as you might expect. In the above example, for instance, a call to SFT2 will bring up rows three and four as defined in the keyset editor. Calls to SFTx, where 'x' is greater than the number of displayable rows available will bring up the last row of labels.

If the terminal has fewer soft keys than are specified in the keyset, each row of the keyset will be split into multiple rows on the terminal, and each row will end with a MORE key as its rightmost key. Blank rows (i.e. those with only a MORE key) will be deleted, and then 2 special rules are applied:

- If there is only one row, the MORE key is removed.
- If there are exactly 2 rows and the second row has only one key plus a MORE key, the two rows are combined into one by discarding the MORE key from each row.

The label that appears on the MORE key will be the same label that appears on any SFTN key in the row. If there is no SFTN key in the row, then the label will be as specified in the message file (usually "MORE").

The rules for SFTN and SFTx stated above apply in this case as well.

If you plan to port your application to multiple platforms, it is a good idea to end each row of soft keys with a MORE key (SFTN). This way, no matter what configuration your terminal or keyset specifies, the user will be able to access all soft keys available. Note that if you change the number of keys available per row in the keyset editor after you have defined a set of keys, the editor will automatically insert a MORE key at the end of each row for you.

Chapter 7
Authoring Reference

This chapter contains useful information for JAM developers. It is arranged alphabetically by topic. The topics and page numbers are listed below:

7.1
COLON PREPROCESSING

Hook string arguments are scanned for *colon variables*. A colon variable is a colon followed by the name of a field, group, or LDB entry (field or group). Each colon variable is replaced with its value. The combination of scanning and replacement is called *colon preprocessing*. Colon preprocessing takes place before the hook string is executed. Colon preprocessing is used extensively in JPL. See the *JPL Guide* for more information.

Consider a JPL procedure named `addinto` that computes the sum of two integers and stores the sum into a variable. It requires three arguments:

- the name of the destination variable,
- the first value to be added, and
- the second value to be added.

The following control string is invoked to store the sum of `op1` (whose content is currently `300`) and `op2` (whose content is currently `500`) into `bigsum`:

```
^jpl addinto bigsum :op1 :op2
```

The colon preprocessor would scan the control string and replace `:op1` and `:op2` with their contents. **JAM** would then execute the resulting control string:

```
^jpl addinto bigsum 300 500
```

To preserve a colon in a hook string, precede the colon with a backslash(\), precede the colon with another colon, or follow the colon with a space. In the first two cases, only the colon remains. In the third case, the colon and space both remain. To cause colon–expanded text to be re–scanned for colon variables, use `:*` instead of a single colon. See the discussion of colon preprocessing in the *JPL Guide* for more information.

The hook strings affected by colon preprocessing are categorized in Figure 75 below. Colon preprocessing doesn't affect the entire hook string, only the portions of the hook string indicated in the table below. Function prototyping is a requirement for colon preprocessing of all hook strings that invoke C functions. This is because such a hook string can have arguments only if it is prototyped.

In the case of prototyped functions, colon expanded arguments whose contents may contain spaces should be enclosed in quotation marks (see Figure 76).

Category	*Subcategory*	*Restrictions*
Control Strings	^ strings	Applies to the arguments (i.e., the entire string after the function name).
	^jpl strings	Applies to the arguments (i.e., the entire string after ^jpl ***procname***).
	! strings	Applies to entire string.
	& strings && strings form strings	Not performed.

Category	*Subcategory*	*Restrictions*
Field, Group, and Screen Function Hook Strings	`jpl` strings	Applies to entire string after `jpl` ***proc-name.***
	strings that invoke C functions	Applies to entire string after function name. The function must be prototyped.

Figure 75: Colon Preprocessing of Hook Strings

The following table shows several hook strings before and after colon preprocessing. Assume that the value of `op1` is `300`, the value of `op2` is `500`, the value of `lumped` is `300 500`, and the value of text is `enter`.

Hook String Before Pre-processing	*Hook String After Pre-processing*
`^jpl add :op1 :op2`	`^jpl add 300 500`
`^jpl add :lumped`	`^jpl add 300 500`
`^jpl :text :op2`	`^jpl :text 500`
`^jpl display ::op1`	`^jpl display :op1`
`^jpl display \:op1`	`^jpl display :op1`
`^jpl display "Err: oops"`	`^jpl display "Err: oops"`
`^myfunc :op1 :op2`	`^myfunc 300 500`
`^:text`	`^:text`
`!:text :op1 :op2`	`!enter 300 500`
`myhook :op1 :op2`	`myhook 300 500`
`myhook ":lumped"`	`myhook 300 500`
`myhook :lumped`	`myhook 300`

Figure 76: Hook String Colon Preprocessing Examples

Remember that colon preprocessing is performed first. Then the resulting string is processed as if it were the original string.

7.2
CONTROL STRINGS

A control string is associated with a function key or with a menu selection field in order to specify window display, form display, JPL execution, C function execution, or program execution. The association of a control string with a function key is done within the Control Strings Screen (see page 82). The association of a control string with a menu selection field is done by placing the control string in the field following the menu selection field, called the menu control field (see page 84 and the Menu section of this chapter). A control string can also be executed directly from JPL procedures with the use of the `call` verb.

The type of action performed by a control string is determined by the leading characters in the control string. The table below summarizes the leading characters and the corresponding actions. The sections that follow explain these actions in detail.

Leading Character	*Action Type*	*Example*
None	Display form.	`mainmenmu`
&	Display stacked window.	`&(5,20)status`
&&	Display sibling window.	`&&(5,20)status`
^	Invoke C function.	`^drop acctno`
^jpl	Invoke JPL procedure.	`^jpl chkcust lname`
!	Invoke program.	`!wordproc :filename`

Figure 77: Leading Characters of Control Strings

7.2.1
Form Control Strings

A form control string has no special leading characters. The first word in the string is interpreted as the name of a screen to be displayed as a form. The screen name may be preceded by viewport parameters (page 152). When a screen is displayed as a form, all open windows (including the base form) are first closed. If the form name is not already on the form stack, then it is added. Otherwise, the form stack is popped until the form name is on top of the stack. See the JAM Overview for a more complete discussion of the form and window stacks.

Several examples of form control strings are presented below:

Form Control String	*Comments*
`mainmenu`	Display `mainmenu` at row 1, column 1 of the physical display.
`(5,20)mainmenu`	Display `mainmenu` at row 5, column 20 of the physical display.
`(1,1,10,40,5,5)mainmenu`	Display `mainmenu` in a 10 row by 40 column viewport positioned at row 1, column 1 of the physical display. Row 5, column 5 of the screen will be initially positioned at the top left corner of the viewport.

Figure 78: Form Control String Examples

7.2.2

Stacked Window Control Strings

A stacked window control string starts with a single ampersand (&). The first word in the string is interpreted as the name of a screen to be displayed as a stacked window. The screen name may be preceded by viewport parameters (page 152). When a screen is displayed as a stacked window, all open windows (including the base form) remain open, and the new window is displayed above the older windows. Stacked windows were the only type of window supported by JAM prior to release 5.

Several examples of stacked window control strings are presented below:

Stacked Window Control String	*Comments*
`&mainmenu`	Display `mainmenu` at row 1, column 1 of the physical display.
`&(5,20)mainmenu`	Display `mainmenu` at row 5, column 20 of the physical display.
`&(1,1,10,40,5,5)mainmenu`	Display `mainmenu` in a 10 row by 25 column viewport positioned at row 1, column 1 of the physical display. Row 5, column 5 of the screen will be initially positioned at the top left corner of the viewport.

Figure 79: Stacked Window Control String Examples

7.2.3

Sibling Window Control Strings

A sibling window control string starts with a double ampersand (&&). The first word in the string is interpreted as the name of a screen to be displayed as a sibling window. The screen name may be preceded by viewport parameters (page 154). When a screen is displayed as a sibling window, all open windows (including the base form) remain open, and the new window is displayed above the older windows. The new window becomes a sibling of the previously active window, and of all windows that are siblings to the previously active window. The user can freely move between sibling windows by pressing the VIEWPORT key (see page 158). The stacked window examples in Figure 79 apply to sibling windows (just add a second ampersand). A window's mode may be converted between stacked and sibling via the library function `sm_sibling`.

7.2.4

C Function Control Strings

A C function control string starts with a caret (^). The first word in the string is interpreted as the name of a C function to be invoked. The C function name may be preceded by a target list (see below). The arguments to the function (but not the function name itself) are processed by the colon preprocessor as described in section 7.1 C functions called from control strings are often referred to as *control functions*.

There are three types of C functions that can be invoked: developer–written, **JAM** library, and built–in. Developer–written functions must be installed with sm_install.

JAM library functions must be prototyped and installed. The built–in functions should not be prototyped or installed. See the *Programmer's Guide* for a discussion of prototyping and installing functions. The *Programmer's Guide* also contains reference pages for the built–in functions; they are summarized briefly below:

- `jm_exit`
  Close active screen and return to previous screen, precisely as if EXIT were pressed.

- `jm_gotop`
  Return to the top level screen, precisely as if SPF1 were pressed.

- `jm_goform`
  Open a window that prompts for the name of a screen to display, precisely as if SPF3 were pressed.

- `jm_keys` ***logical_key_or_string_list***
  Place the specified **JAM** logical keys on the keyboard input queue, to be processed by **JAM** as if each logical key were pressed in order. For example, "`^jm_keys EXIT CLR`" makes **JAM** behave as if EXIT and CLR were pressed in succession; "`^jm_keys HOME 'hello'`" will cause **JAM** to act as if the HOME key were pressed and then the word 'hello' was typed.

- `jm_mnutogl`
  Toggle the active screen between menu mode and data entry mode, precisely as if MTGL were pressed.

- `jm_system`
  Open a window that prompts for a program to be executed by the operating system, precisely as if SPF2 were pressed.

Several C function control strings are presented below:

C Function Control String	*Comments*
`^drop acctno`	
`^verify :name :idnum`	Invoke `verify`, passing the contents of `name` and `idnum` as arguments.
`^jm_exit`	Invoke the built–in function `jm_exit` to simulate the default action of the EXIT key.
`^jm_keys CLR HOME`	Invoke the built–in function `jm_keys` to simulate pressing CLR followed by HOME (clear the screen and move the cursor to the home position).

Figure 80: C Function Control String Examples

The C function must return an integer. If the integer corresponds to the value of a **JAM** logical function key, then the **JAM** Executive processes that key. For example, if a function returns PF4, then **JAM** behaves as if PF4 had been pressed by the user. The function should return 0 if there is no key to process.

Target Lists

A target list is a list of function return values associated with control strings. As discussed above, the value returned by a control function is normally interpreted as a logical function key to be processed by the **JAM** Executive. A target list provides an alternative mechanism for controlling the actions to be taken as a result of the execution of a control function or of a JPL procedure (JPL procedure control strings are discussed below). It enables conditional execution of multiple control strings.

For example, the following control string invokes the function named `inquire`. If `inquire` returns –1, then the built–in function `jm_exit` is invoked. Otherwise, nothing happens (i.e. the active screen remains active):

```
^(-1 = ^jm_exit)inquire
```

Syntactically, a target list appears, in parentheses, after the caret and before the function name. The list consists of semi–colon separated pairs of the form:

return_value = ***control_string***

where ***return_value*** is a decimal or hexadecimal (specified by starting with `0x`) integer returned by the function. The ***control_string*** will be executed when the actual return value matches ***return_value***. If the target item contains only a control string, then that control string is executed by default when none of the preceding return values are matched. Since the pairs are evaluated from left to right, the default should be the last item in the target list.

Target lists may be nested to any depth. That is, a control string in a target list may contain its own target list. The following example illustrates nesting, multiple target items, and a default item:

```
^(-1=^(^jm_exit)cleanup; 1=&success; &&(5,5)defwin)process
```

The control string first invokes the C function named `process`. If `process` returns -1, then `cleanup` is invoked. The built-in function `jm_exit` is invoked when `cleanup` returns, regardless of the value returned by `cleanup`. If `process` returns 1, then the screen `success` is displayed as a window. If `process` returns any other value, then the screen `defwin` is displayed as a sibling window at row 5, column 5 of the physical display.

Colon preprocessing is performed on control strings in a target list (not on the return value) after the control string's return value is matched. Therefore, a nested control string with a colon–expanded argument can be impacted by changes to the the value of the colon variable by a C function called earlier within the same control string. For example, in the following control string, the value of `uname` passed to `ident` can be changed by the function `lookup`:

```
^(^ident :uname)lookup
```

7.2.5

JPL Procedure Control Strings

A JPL procedure control string starts with a caret followed by the word `jpl` (ie. – `^jpl`). The second word in the string is interpreted as the name of a JPL procedure to be invoked. The procedure name may be preceded by a target list (see above). The arguments to the function (but not the function name itself) are processed by the colon preprocessor as described in section 7.1

Several JPL procedure control strings are presented below:

JPL Procedure Control String	*Comments*
`^jpl chkcust lname`	
`^jpl find :lname`	Invoke `find`, passing the contents of `lname` as an argument.

Figure 81: JPL Procedure Control String Examples

The JPL procedure must return an integer. If the integer corresponds to the value of a JAM logical function key, then, in the absence of a target list, the JAM Executive processes that key. For example, if a procedure returns PF4, then JAM behaves as if PF4

had been pressed by the user. The procedure should return 0 if there is no key to process.

7.2.6
Program Control Strings

A program control string starts with an exclamation point (!). The entire string following the exclamation point is interpreted as a program and its arguments to be executed by the operating system. At runtime, the string (program name and arguments) are colon–preprocessed and passed to the operating system for execution. After the program execution is complete, the user is prompted to press the space bar before returning to JAM.

Several examples of program control strings are presented below:

Program Control String	*Comments*
`!wordproc :filename`	Invoke the word processor to edit the file whose name is stored in the variable `filename`.
`!dir`	Display a directory listing under some operating systems.

Figure 82: Program Control String Examples

7.3
HELP SCREENS

JAM's help screen feature associates a help window with a field or an entire screen. To associate a help screen with a field or screen, follow the instructions on pages 33 and 53. Each help screen is created with the Screen Editor. The purpose of this section is to describe the several ways in which help screens can function and to describe how to construct help screens. The following types of help screens are described in this section:

- Help Screen With Display Data Only.
- Help Screen Containing A Menu
- Help Screen With Data Entry Fields

- Help Screen With Field–Level Help Sub–Screens

In all cases, a help screen may have its own screen–level help screen associated with it. In addition, as of **JAM** release 5, function keys associated with control strings work on help screens.

7.3.1

Help Screen With Display Data Only

The simplest help screens, like the screen shown below, are display–only. The screen may contain fields with protection from data entry and tabbing into, possibly filled from the LDB or from a screen entry function. The help window is closed when EXIT is pressed.

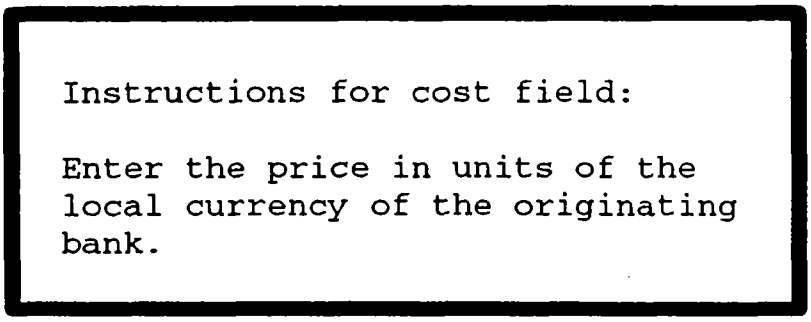


Figure 83: Display–Only Help Screen

Any help screen with no unprotected fields or menu fields will be treated as display only. To create a scrolling help screen, create a scrolling array with a word–wrap edit, and protect it from data–entry and clearing.

7.3.2

Help Screen Containing A Menu

A help screen can contain a menu that calls up additional help screens, as in the help screen shown below.

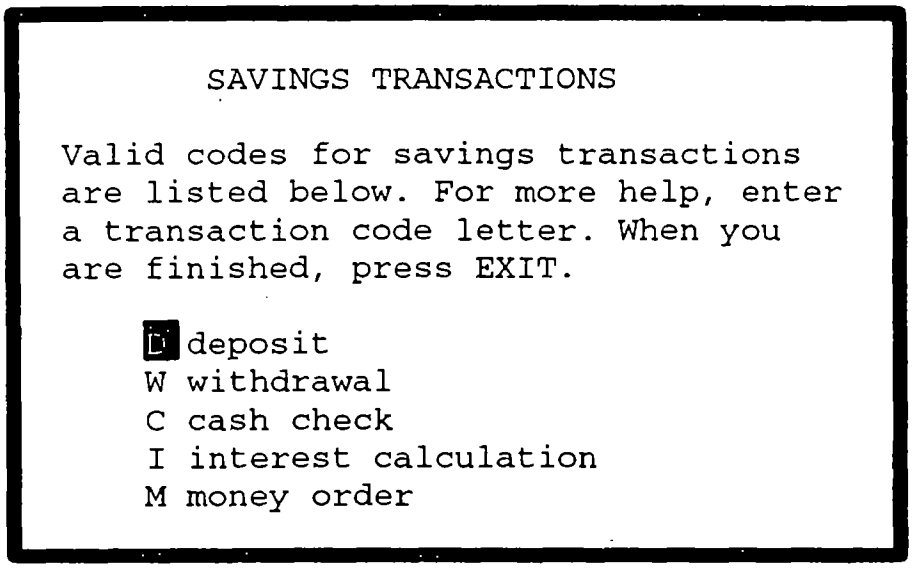


Figure 84: Help Screen Containing A Menu

To create a help screen with a menu:

- Create a screen with fields that have the menu field edit (see page 46). Don't create menu control fields; they are ignored during help screen processing.
- For each menu field, use the help screen field attachment (page 53) to specify an associated lower–level help screen.

When the user makes a selection from the menu, the associated lower–level help screen will appear.

7.3.3

Help Screen With Data Entry Fields

In some cases, it may be desirable for the user to be able to enter data into a screen while the associated help screen is displayed. This can be done if the help screen is created with a field for data entry.

When the help screen is displayed, the content of the associated field is copied into the field on the help screen. The user may then enter data into the help screen field. When the user presses XMIT, the content of the help screen field is copied back into the associated field, and the help screen closes. If the user presses EXIT, then the field is not copied.

The help function provides for data entry automatically whenever the help screen contains exactly one field that is not protected from data entry and tabbing into. The data

entry field on the help screen is normally defined to be the same length as the associated field. If the data entry field is too short, then it is automatically made shiftable, with a maximum length equal to the length of the associated field. If it is too long, then it is shortened. In addition, the following edits are copied to the field on the help screen:

- character
- right justified
- upper or lower case
- data required
- must fill
- clear on input
- check digit
- currency format
- range check

7.3.4
Help Screen With Field–Level Help Sub–Screens

The help function will not process both data entry and menu fields on the same screen. However, it is possible to provide additional help screens for a data entry help screen by associating help with the help screen data entry field, or by associating help with protected fields.

The screen shown below is an example of the use of the latter feature.

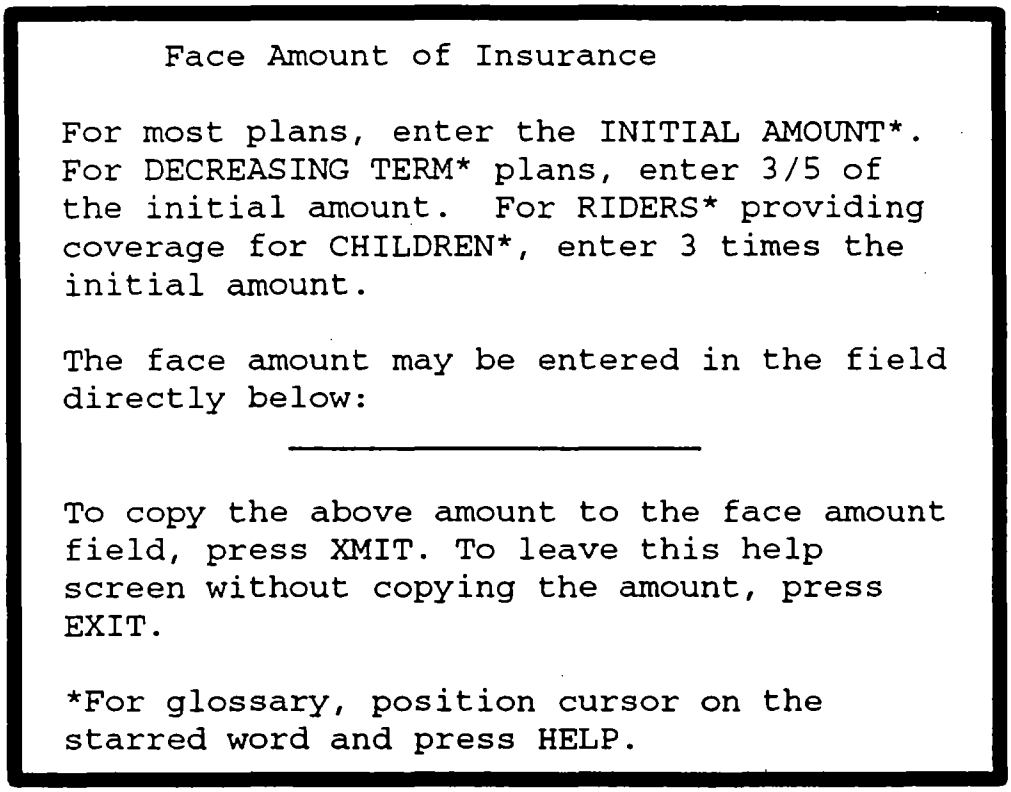


Figure 85: Help Screen With Field–Level Help Sub–Screens

When the cursor is positioned on one of the keywords (shown in capitalized text) and the HELP key is pressed, the associated lower–level help screen is displayed. To create a help screen using this feature:

- Create a help screen with all text entered as display data.
- Replace each keyword by underscores, and press XMIT to convert the underscores into fields.
- Re–enter the keywords into the fields. Make the keywords distinguishable from the rest of the text by using display attributes, or by using other means such as capitalization.
- Protect each keyword field from data entry, but not from tabbing, and assign it a help screen (page 53).

Item selection screens may also be viewed as help screens. Item selection is described on page 54.

7.4 LOCAL DATA BLOCK

The purpose of this section is to discuss the interaction between entries in the LDB and fields and groups on a screen. The LDB is created from the data dictionary (DD) at run-time (and when the authoring tool is entered in application mode). The DD editor is discussed in chapter 5 on page 99. Access to the DD from the Screen Editor is discussed in the Screen Editor chapter in sections 4.3.12 (Data Dictionary Search) and 4.3.13 (Add to Data Dictionary) on pages 89 and 91. LDB initialization is discussed on page 109. Manipulating the LDB entries by scope is discussed on page 104.

The LDB is a table of entry names with associated values. Each entry is either a field entry or a group entry. Each entry has a maximum number of occurrences and a length, as defined by the corresponding entry in the DD. Field entries also have field characteristics needed for proper formatting of data, such as the currency format. The values in the LDB are maintained for the duration of the application; they do not disappear when a screen is closed. Therefore, the LDB can be used to move data between screens and to preserve data that is displayed only on a single screen after the screen has been closed. Most of the discussion below applies equally to field and group entries; for convenience we will generally use the term field to mean a field or a group entry.

Screen fields and LDB entries are matched by name. If the field name is the name of a field entry in the LDB, then the field is said to be in the LDB. When a user enters data into a field that is in the LDB, then the data is copied to the LDB before any other screen is displayed. When a screen containing fields in the LDB is displayed, the fields are filled with the content of the corresponding LDB entry. However, initial data in a field (data entered into the field with the Screen Editor) overrides the content from the LDB. Note that truncation occurs when the length of the destination (screen field or LDB entry) is less than the length of the source (screen field or LDB entry). If the maximum number of occurrences in the destination (i.e. the number returned by the library function `sm_max_occurs`) is less than the allocated number of occurrences in the source, then data is copied until the destination is full. In all cases, the allocated number of occurrences of the destination is set to the number of occurrences actually copied.

JAM library functions that access field values by field name (e.g. `sm_n_getfield`) seek them first on the screen, and then in the LDB, except at screen entry, when the order is reversed. Math calculations and JPL procedures work in the same fashion. This preserves the illusion that fields in the LDB are always accessible and always have the latest values in them. However, functions, math calculations, and JPL procedures that refer to fields by number do not search the LDB — LDB entries do not have field numbers.

One important characteristic of a group entry is that its value is the list of selected group occurrence numbers. Each selected occurrence number is stored in one occurrence of

the group entry. The value 0 indicates the end of the list (no entry will contain a 0 if all occurrences are selected). Placing a group in the LDB causes the same group on different screens to have the same set of selected fields. However, a group entry in the LDB does not hold the contents of the fields in the group. It is the developer's responsibility to ensure that two groups linked through the LDB have the same number of fields, in the same order, with the same contents. This can be accomplished most easily by placing the group's fields in the LDB and by using the Screen Editor clipboard to copy the group from screen to screen during development.

As is true of the DD, a group entry and a field entry cannot share the same name in the LDB. However, the matching of screen fields to LDB entries and screen groups to LDB entries is done without regard to whether the LDB entry is intended to be a field or group entry.

7.5
MENUS

The purpose of this section is to describe how to build the types of menus that can be built with **JAM**. The following topics are discussed elsewhere:

assigning the menu field edit .... page 46
submenus ..... page 46
shortcut menu creation ..... page 84

A screen can behave both as a menu and as a data entry screen, but not at the same time; menus are active only when the screen is in *menu mode*, while data entry is possible only in *data entry mode*. When a screen has menu fields and no other unprotected fields, it is always in menu mode. When a screen has no menu fields, it is always in data entry mode. However, if a screen has both menu fields and unprotected data entry fields,the end user can toggle the mode by using the MENU TOGGLE key, or by clicking the mouse in one type of field or the other. On these mixed use screens, **JAM**'s default is to start the screen in data entry mode. To force a screen to start in menu mode, modify the screen's characteristics as described on page 33.

Menus that are processed by the **JAM** Executive have one pair of fields for each menu item: a menu selection field and a menu control field. The menu selection field contains the text of the menu selection; *it must also have the menu field edit assigned to it* (see page 46). If the screen will also be used in data entry mode, then the selection field should be protected from tabbing into. The menu control field contains the control string (see page 124) associated with the menu item. The control field must have a field number that is one greater than the field number of the associated selection field. The easiest way to do that is to place the control field on the same line as the selection field

and immediately to its right. The control field should be made non–display. To minimize the use of screen real estate, the control field can be made into a shifting field with as few as one onscreen character position. An example menu is shown below in Figure 86 as it would appear in DRAW mode with its control fields visible. It is shown again in Figure 87 as it would appear in TEST mode (or application mode or at run-time), with a bounce bar indicating the current choice. The bounce bar appears in toggled reverse video, so if a menu field already had the reverse–video edit, the bounce bar would appear in non–reverse video. Note that a menu will have a bounce bar in TEST mode only if the screen is in menu mode (as opposed to data entry mode).

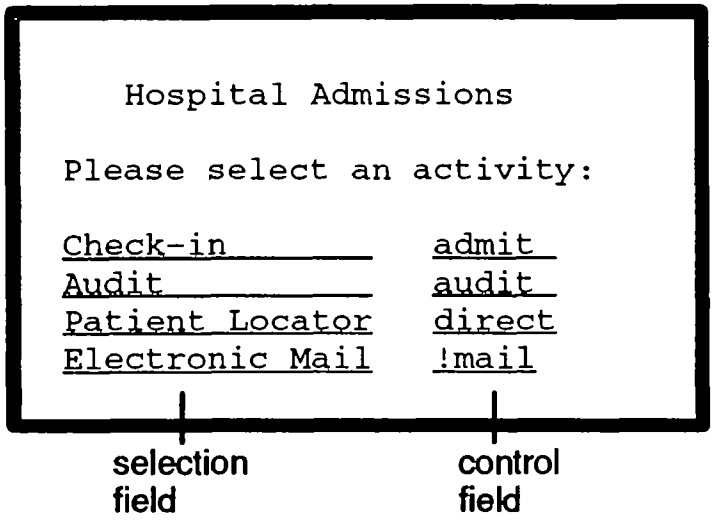


Figure 86: Menu Screen In Draw Mode

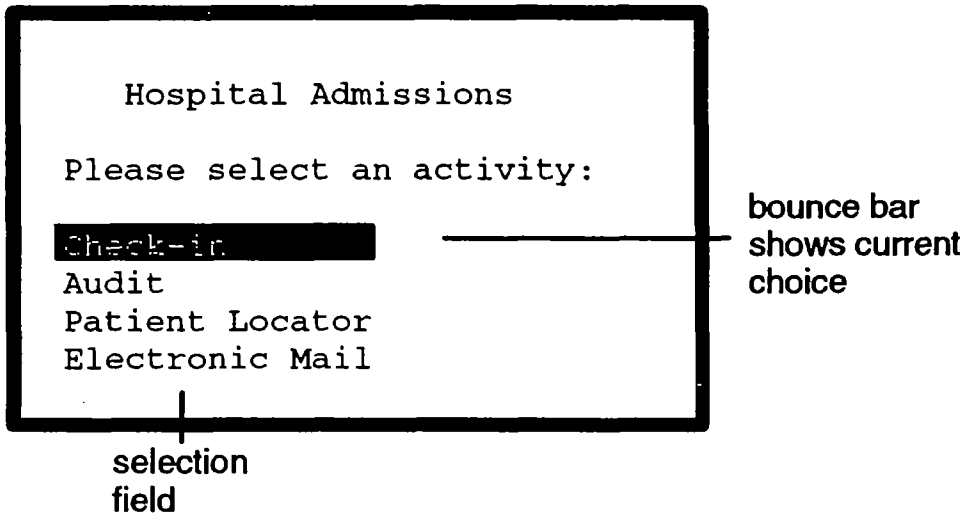


Figure 87: Menu Screen In Test Mode

A user moves the bounce bar with the TAB, BACKTAB, BACKSPACE, arrow, and space bar keys. Pressing XMIT or NL selects the current choice, and thereby executes the control string found in the corresponding control field. Alternatively, typing the first character of a selection field (or multiple characters if more than one field starts with the same character) causes that menu item to be selected. If more than one character is required, then the bounce bar moves to the next possible selection as characters are typed. Note that if all menu selection fields are wholly upper case or wholly lower case, then **JAM** ignores case, so that `a` and `A` are equivalent.

The processing of menus may be changed by the library function `sm_option`. For example, the selection can be based on the first upper case character of a choice rather than the first character (as in the Screen Editor's Color Menu shown on page 25).

While many menus consist of a vertical list of options (Figure 87), **JAM** permits any organization for menus, including horizontal menus (Figure 88) or more exotic menus, such as Figure 89 , in which the menu selections are the headings of introductory paragraphs. These other menus work because **JAM** considers all non-blank fields with the menu field attribute to be menu selection fields.

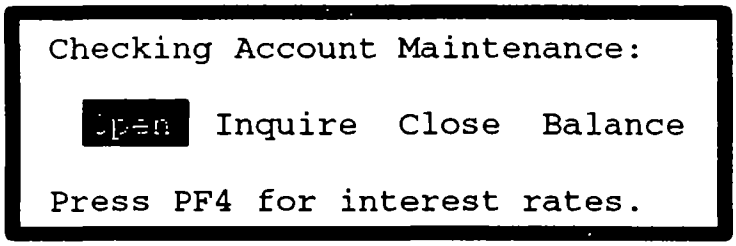


Figure 88: Horizontal Menu

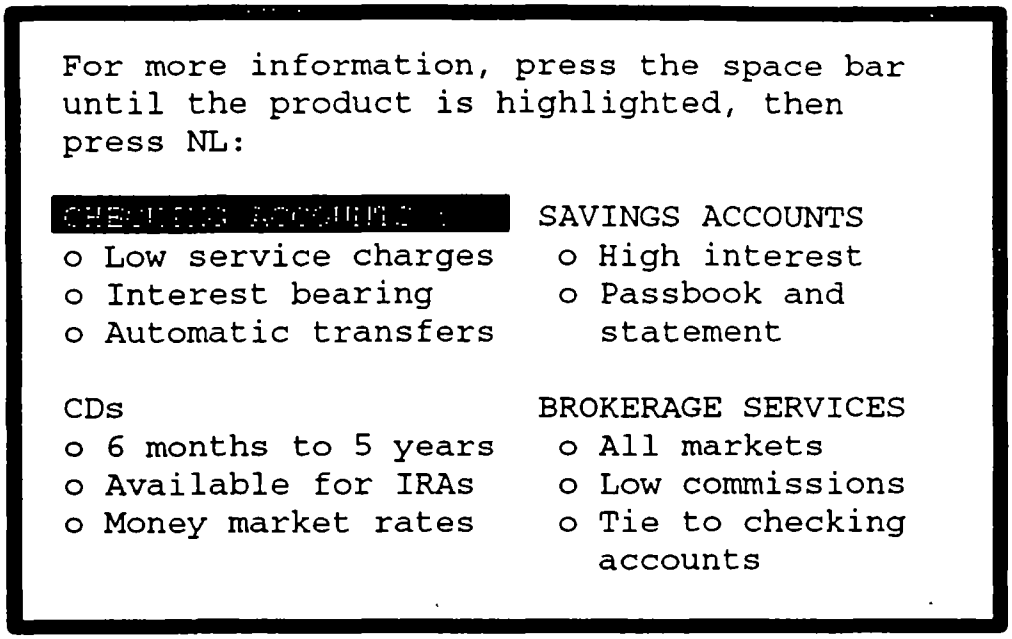


Figure 89: Topical Menu

7.5.1

Dynamic Menus

It is sometimes desirable to create a menu whose content changes as circumstances dictate. For example, security considerations might require that a user be able to see and execute authorized menu options only. This is possible because **JAM** never considers blank fields to be menu selection fields. Therefore, a dynamic menu may be created by performing the following steps:

- Create a screen with enough menu edit fields to hold the maximum possible number of menu selection fields. Also create corresponding menu control fields.
- Fill the selection and control fields at runtime from the LDB or with a screen entry function. Leave unneeded menu edit fields blank. If you wish to always have the same choices appear at the same position on the screen, then leave blank fields as placeholders rather than placing all blank fields at the end of the screen.
- If you wish to shrink the screen to fit around the resulting menu, then call the library function `sm_shrink_to_fit` after the menu fields have been populated.

7.6
REGULAR EXPRESSIONS

A regular expression is a pattern or template made up of characters. It divides ordinary character strings into two kinds: those that match the pattern, and those that don't.

JAM supports regular expressions in the style of the UNIX editors, and uses them to check that the contents of a field conform to a pattern. The pattern can be defined in a way that is extremely flexible, and unlike most **JAM** edits can be used to restrict different parts of the field to different character types, or classes.

Regular expressions can be installed as character edits or field edits. When installed as a character edit, the input is matched one character at a time with the regular expression and invalid input is rejected immediately. When installed as a field edit, the entire field is matched against the regular expression when the field is validated. Character level regular expressions are also verified at field validation in case the user has deleted a character, causing the string to become invalid.

When **JAM** checks a field against a regular expression, it steps through the field data and the regular expression together. It matches as many field characters as it can against the first subexpression before going on to the next, and quits at the first mismatch.

Here is an example of a regular expression. This one defines a sort of ID number that is three digits, followed by a dash, followed by at least three letters or numbers, possibly more, up to the length of the field:

```
[0-9]\{3\}-[a-zA-Z0-9]\{3,\}
```

7.6.1
Forming Regular Expressions

There are two kinds of rules for constructing a regular expression. One kind tells you how to form a simple expression, and the other tells you how to combine expressions into more complex expressions. The basics of regular expressions are quite simple; however, by combining them, you can quickly arrive at expressions that are quite complex. The following discussion, therefore, proceeds from simple rules for forming simple expressions to more complicated rules for combining and repeating expressions.

Simple Expressions

The simplest regular expression is a single character, which matches itself. The regular expression `z` matches the string "z". There are only a few characters that are special and do not match themselves; they are explained in the ensuing discussion.

Blanks are not special, but they are not ignored either. A blank in a regular expression matches a blank in a field.

A dot (.) is a special character; it matches any single character, including (but not limited to) itself. The backslash (\) is also special. It is a quote character: it forces the following character to match itself even if that character is special. For instance, the sequence \. matches the dot ".", and the sequence \\ matches a single backslash.

The backslash also makes certain ordinary characters special. Examples of these are the quoted braces and quoted parentheses used for repeat counts and grouping respectively. See the following sections for details.

Character Classes

A group of characters between brackets ([]) matches a single occurrence of any of the characters. `[13579]` matches any odd digit, and `[aA]` matches an `a` of either case. The group of characters is called a character class. The order of the characters in the class is not significant. In fact, when **JAM** saves a regular expression, it sorts the characters in the character class by ASCII value.

Long lists of consecutive characters can be abbreviated using a hyphen (-). For instance, `[a-z]` matches any lowercase letter, and `[A-Za-z]` matches any letter at all. Because of the ASCII collating sequence, `[A-z]` matches all letters *plus* some punctuation characters that fall between `Z` and `a`. You may use any number and combination of characters and ranges within one set of brackets.

Character classes can be negated by placing a caret (^) immediately after the opening bracket in the class. This will cause the class to match any character not in the class. The expression `[^0-9+.-]` matches any non–numeric character.

Special Characters in Character Classes

NOTE: *Only* the caret, the hyphen, and the closing bracket are special characters within a character class. The backslash is not a special character and cannot be used as the quote character in a character class. It stands for itself.

Listed below are the exceptions, when a carat, hyphen or bracket stands for itself:

- The caret stands for itself if it *is not* the first character.
- The closing bracket stands for itself if it *is* the first character, or if it is the first character after an opening caret.
- The hyphen stands for itself if it *is* the first or last character, or if it is the first character after an opening caret..

For example, to accept any character *except* ^, -, [,], ., or \, use the following:

```
[^][.\^-]
```

Concatenating Subexpressions

The simplest way of combining two or more expressions is to put one after another. They then match whatever matches the first, followed by whatever matches the next, and so on. The expression `JYACC` matches the string "JYACC", the expression `a[0-9]` matches an `a` followed by a digit.

Repeating Subexpressions

The asterisk (`*`) causes the preceding subexpression to match zero or more characters that match the subexpression, instead of only one character. The expression `[0-9]*` matches any number of digits or none at all. To force at least one digit, the expression `[0-9][0-9]*` is used.

A more definite repeat count for an expression can be specified by enclosing the count in quoted curly braces `\{` and `\}`. The repeat count follows the subexpression to be repeated, and takes one of the following three forms:

- `\{n\}` exactly `n` repetitions
- `\{n,\}` `n` or more repetitions
- `\{n,m\}` at least `n` repetitions but no more than `m`

For example, `[0-9]\{5\}` matches a numeral of exactly five digits, or an old–style zip code.

Re–matching Subexpressions

To re–match an expression or sequence of expressions, use quoted parentheses `\(` and `\)` around them. If you place a quoted number later in your expression, say `\n`, it will match whatever the nth subexpression surrounded by `\( \)` matched. Note that the subexpression is not reproduced, but the actual character data is. The expression `\([0-9]*\)\.\1` matches `123.123`, or any other real number where the integer and fractional parts are the same, but it will not match `123.45`.

7.6.2

Regular Expression Examples

Some examples of regular expressions follow below:

`[iI][cC][eE]` matches `ice, icE, iCe, Ice, IcE, ICe, or ICE.`

`212-[0-9]\{3\}-[0-9]\{4\}` matches any phone number in the Manhattan or Bronx boroughs of New York City.

`[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}` matches any social security number.

`[a-zA-Z_][0-9a-zA-Z_]*` matches an identifier in the C programming language.

7.6.3

Summary of Special Characters In Regular Expressions

Character	*Name*	*Function*
\	backslash	makes any special character, including itself, ordinary. Makes { } ( ) and digits special in certain contexts.
.	dot	matches any single character.
[ ]	square brackets	surround a character class expression.
*	asterisk	causes the preceding character class or single character expression to match zero or more occurrences.
\(\)	quoted parentheses	surround an arbitrary subexpression for the purpose of re–matching.
\1	quoted digits	re–match a previous expression enclosed in quoted parentheses \(\)
\{ \}	quoted curly braces	surround a repeat count to the preceding character class or single character expression.

Figure 90: Special Characters (not within a Character Class)

Character	*Name*	*Function*
^	caret	negates the character class when it immediately follows [
–	hyphen	denotes a character class range unless it is the first or last character in class.
]	square brackets	closes character class unless it is the first character, or the first character after an opening ^

Figure 91: Special Characters within a Character Class

The caret (^) and dollar sign ($), which represent the start and end of a line respectively in some UNIX utilities, do not have that special meaning within JAM regular expressions.

7.7
SCROLLING ARRAYS

The purpose of this section is to explain the behavior of scrolling arrays, particularly in response to keyboard entry. The first section discusses single scrolling arrays. The second section discusses synchronized scrolling arrays. There are several sections of the Screen Editor chapter that should be read prior to reading this section:

1. The description of scrolling and simple arrays, maximum and allocated array size, and automatic synchronization of arrays that is discussed with respect to the Field Size Screen, starting on page 69.
2. The description of how to synchronize arrays manually, starting on page 95.

7.7.1
Single Scrolling Arrays

The capacity of an array is measured in terms of the maximum number of occurrences that it can contain. However, it is often desirable for an array to behave as if it contained fewer occurrences. Consider a scrolling array named `friends` with three elements and a maximum of 100 occurrences (hopefully this is not too optimistic), shown in Figure 92 below. The array begins with three allocated occurrences, since **JAM** always allocates at least enough occurrences to hold the onscreen occurrences (elements).

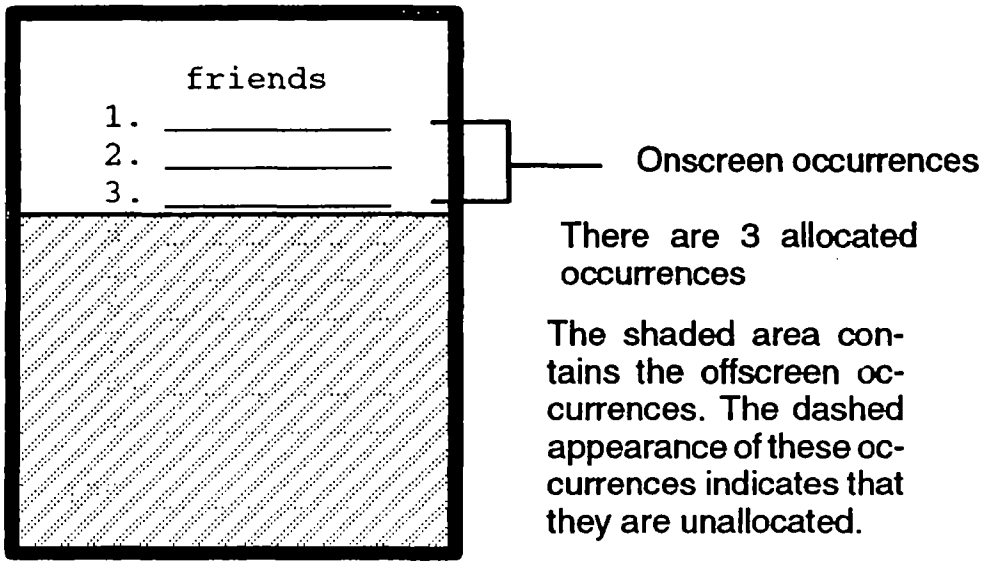


Figure 92: The `friends` Scrolling Array

Cursor movement keys move among the allocated occurrences of a scrolling array, leaving the array only when the last (or first) allocated occurrence is reached. Cursor movement keys never move the cursor into unallocated occurrences. If the user presses PAGE DOWN within the friends array as shown in Figure 92, **JAM** displays a message that indicates that no more data may be viewed. This behavior prevents the user from needlessly scrolling through the often large number of blank occurrences at the end of an array. If the user presses the DOWN ARROW key with the cursor in the third occurrence of `friends`, then the cursor moves to the next field on the screen (or to the first occurrence of `friends` if it is the only unprotected field on the screen or if `friends` is a circular array).

Consider the partially filled `friends` array shown in Figure 93 below.

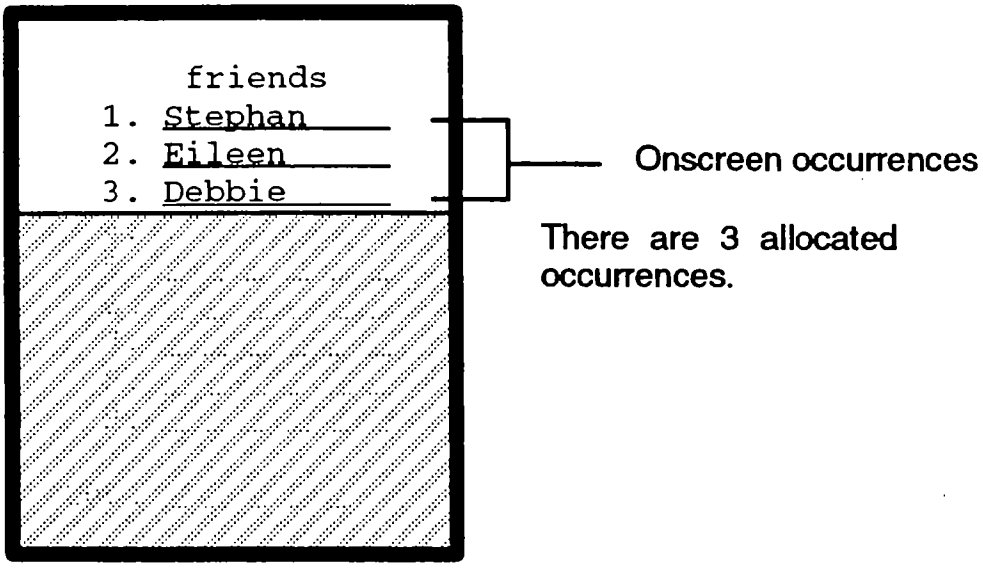


Figure 93: The array partially filled.

The user now has three friends entered, and would like to enter a fourth friend. Pressing DOWN ARROW won't work. The solution is to press the NL key. If the cursor is on the last allocated occurrence of a scrolling array, and if more occurrences can be allocated (ie.—the number of allocated occurrences is less than the maximum number of occurrences), then NL will allocate a new occurrence and scroll the array so that the cursor is positioned in the newly allocated occurrence, as shown in Figure 94.

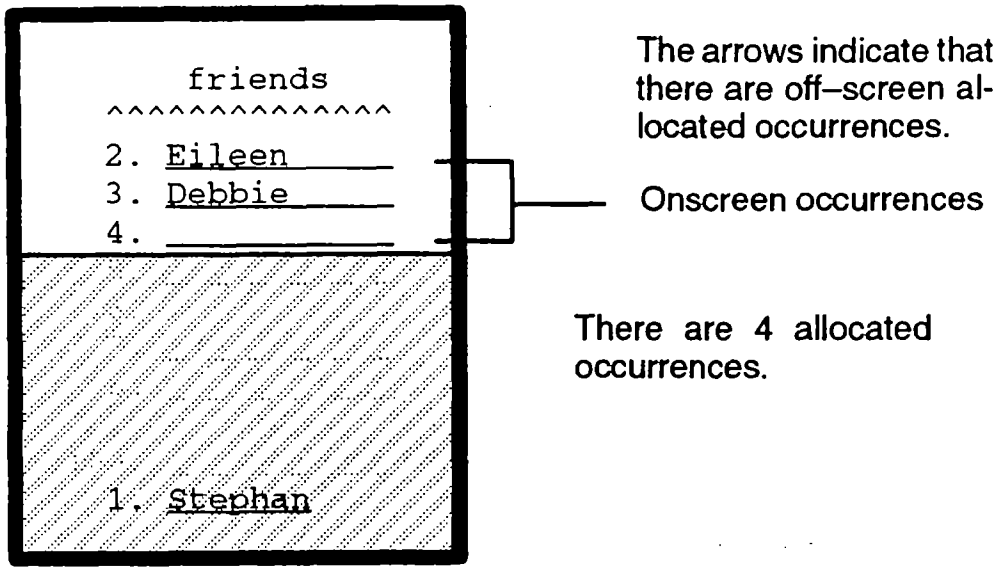


Figure 94: `friends` With A Newly Allocated Occurrence

Although TAB doesn't normally cause new occurrences to be allocated, it can if the "next field" edit is used. For example, TAB behaves exactly like the NL key (until the maximum occurrence number is reached) if the next field specified for `friends` is `friends[+1]`. Please see the more detailed example in the Screen Editor chapter that is associated with Figure 31 (page 52). Two other special cases where an occurrence can be allocated without pressing NL, are if the field has AUTOTAB enabled, and it is filled, or if the array is a "word wrap" array, and a field is filled.

Once an occurrence is allocated, all cursor movement keys can be used to scroll the array to enter data into the occurrence. Therefore, pressing the UP ARROW key with the cursor on `Eileen` in Figure 94 scrolls the array and moves `Stephan` to the top of the screen, as shown below in Figure 95 (note that the fourth occurrence is still allocated). Following the UP ARROW with three DOWN ARROWs scrolls the array back to how it looks in Figure 94. A subsequent DOWN ARROW moves the cursor to the next field on the screen (or back to `Eileen` if `friends` is the only unprotected field on the screen) without scrolling the array.

Note that in arrays with a distance between elements that is greater than 1, it is possible to place a field between two consecutive elements of an array. In such a case, the UP ARROW and DOWN ARROW keys will move among the occurrences of the array rather than into the field as long as the array is a scrolling array. This is consistent with the way arrow keys are defined for scrolling arrays.

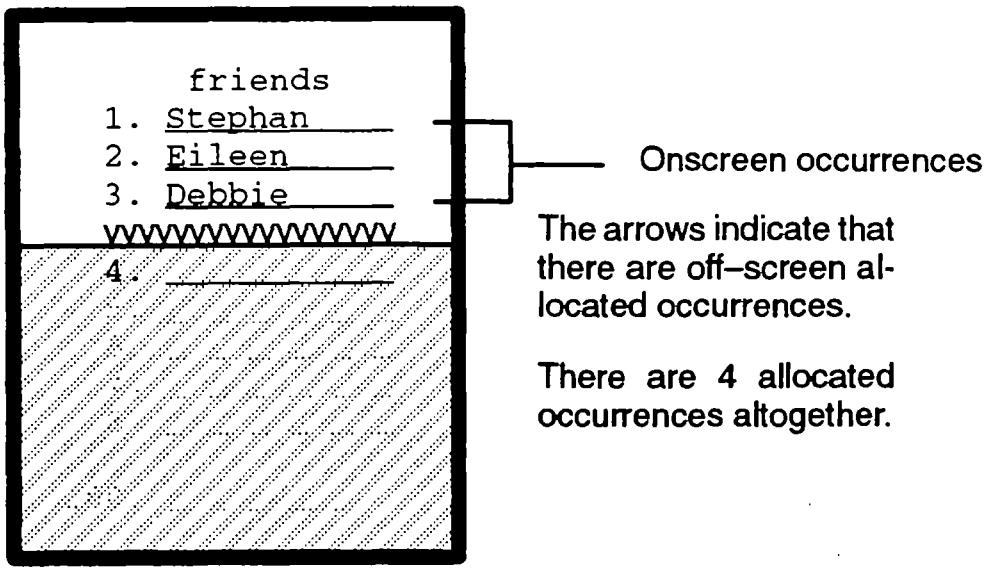


Figure 95: A Blank Offscreen Allocated Occurrence

The data required edit (page 41) is affected by the number of allocated occurrences. It requires that each allocated occurrence have data. Therefore, none of the above example `friends` screens except Figure 93 would pass the data required edit, as they each have at least one blank allocated occurrence.

Occurrences can also be allocated with the INSERT LINE key and by writing data into an un–allocated occurrence with a library function (such as `sm_i_putfield`), with JPL (e.g. `cat friends[99] Robert`), or with a math edit calculation. The INSERT LINE key moves the data in the occurrence beneath the cursor, and in all subsequent occurrences, down one occurrence and allocates an occurrence for the highest numbered occurrence moved (subject to not exceeding the maximum number of occurrences). INSERT LINE does not cause a new occurrence to be allocated if it is just pushing an empty occurrence off the screen. In addition, INSERT LINE does not work in an array that is protected from clearing (otherwise it would be impossible to maintain a synchronized list of numbered items — see the example associated with Figure 96 on page 149.)

Occurrences (in an array that is not protected from clearing) can be de–allocated with the DELETE LINE and CLR keys — except that the first N occurrences, where N is the number of onscreen occurrences (elements), can never be de–allocated. FIELD ERASE and the space bar cannot de–allocate an occurrence. The DELETE LINE key deletes the data in the occurrence beneath the cursor, moves the data in subsequent occurrences up one occurrence, and de–allocates the last allocated occurrence, unless that occurrence is onscreen or protected from clearing. Using DELETE LINE to de–allocate an occurrence does not affect the validation state of the items in the array. The CLR key de–allocates all allocated occurrences.

Occurrences that are allocated in arrays with embedded punctuation are treated specially when allocated. When a new array occurrence is allocated, embedded punctuation is copied from the first occurrence of the array to the newly allocated occurrence. Please refer to the Screen Editor chapter (page 39) for a discussion of embedded punctuation.

7.7.2

Synchronized Arrays

The behavior of synchronized arrays is largely determined by the following rules enforced by **JAM**:

1. If pressing a key would cause a synchronized array to scroll if it were unsynchronized, then pressing that key will cause the array to scroll along with all of its synchronized arrays.

2. Each array in a set of synchronized arrays always has the same number of allocated occurrences as every other member of the set. This means that if a key press allocates (de–allocates) and occurrence of an array, then the same occurrence is allocated (de–allocated) for every array synchronized with that array. A consequence of this rule is that if **JAM** cannot de–allocate an occurrence from one synchronized array, perhaps because the array is protected from clearing, then **JAM** will not de–allocate that occurrence from any synchronized array. The usefulness of this consequence is shown below by example.

Consider the `newfriends` screen shown below in Figure 96. It is identical to the `friends` screen, except that it has two additional arrays, both synchronized with `friends`: `frnum` and `frsex`. `frnum` is fully protected, `friends` and `frsex` are unprotected. Note that the original `friends` screen had numbers on it, but they were present only to show the occurrence number. The purpose of `frnum` is to make it easy to count the number of friends and to keep track of the position of the cursor within the list of friends.

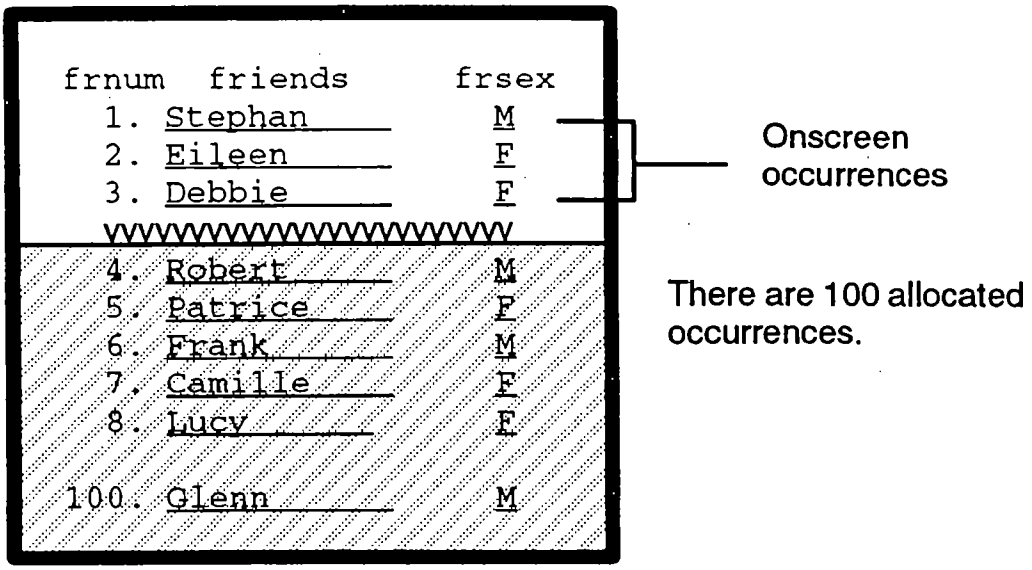


Figure 96: `newfriends` With Synchronized Arrays

The cursor movement keys will scroll all three arrays through all allocated occurrences — which is all 100 occurrences. Every occurrence of `frnum` contains data, and is therefore allocated. Since all synchronized arrays must have the same number of allocated occurrences, all three arrays must have 100 occurrences. Let's move `Frank` up on the list of friends, just behind `Eileen`. The following keystrokes will accomplish the task:

1. Press DOWN ARROW five times to position the cursor on top of `Frank`.

2. Press DELETE LINE to delete `Frank`. Since `frnum` is protected from clearing, it does not change (we don't want a gap in our number sequence) and therefore the number of allocated occurrences in all three arrays remains at 100. The data in the `friends` array is moved up to replace `Frank`. The data in the `frsex` array is moved up to remain associated with the proper friends.

3. Press UP ARROW three times to position the cursor on top of `Debbie`.

4. Press INSERT LINE to insert a blank line between `Eileen` and `Debbie`. `frnum` still doesn't change, since it is protected from clearing, but the data in the other two arrays move down.

5. Re-enter `Frank` into `friends` and `M` into `frsex`.

7.8 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 once the screen is displayed, but it can be cleared with the library function `sm_bitop`.

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.

7.8.1 Fields That Are Not Part of a Group

This section contains information about validation that pertains to fields that are not part of a group. The edits that are part of the validation process for a field and that must be fulfilled before the field's validation bit is set are:

- field edits
- date/time format
- check digit
- currency format
- field validation function (C or JPL)
- field–level JPL procedure

In addition, the math calculation associated with a field is performed as part of the validation process. If all of the above edits are fulfilled, then the field's validation bit is set. The order of processing, which stops as soon as an edit fails, is:

1. The field edits, date/time format, and check digit.
2. The math calculation.
3. The currency format.
4. The field validation function.
5. The field–level JPL procedure.

Field validation occurs when the following events occur, and only for fields that are not protected from validation (page 41):

1. The cursor leaves the field due to a TAB (or NL) or an auto–tab. Normally, the arrow keys do not cause validation. The library function `sm_option` can be used to cause validation to occur whenever the cursor leaves a field. Be careful in the case of cross–field validations; e.g., consider a screen with a country field and a city field. If the user enters France into the country field, and then Rome into the city field, the validation function would reject Rome and then prevent the user from moving the cursor to the country field in order to change the country to Italy. An individual field can also be validated by calling the library function `sm_fval`.
2. The XMIT key is pressed. All fields on the screen are validated. This occurs whether or not there is a control string associated with XMIT. Note that XMIT is the only key that causes all fields to be validated. Other keys can perform the same function if specified by a call to the library function `sm_keyoption`. All of the fields on a screen can be validated by a call to the library function `sm_s_val`.

7.8.2
Fields That Are Part of a Menu or Group

This section contains information about validation that pertains to fields that are part of a menu or group. Validation does not occur when a menu or group field is exited by tabbing or when the XMIT key is pressed. For menu fields and radio buttons, it occurs only when the field is selected. For checklists, it occurs when the field is selected or de–selected with a key, such as NL or the first letter of the content of the field. Note that the library functions sm_select and sm_deselect do not cause a group field's validation function to be executed.

7.8.3
Group Validation

This section contains information about validation that pertains to groups. The validation process for a group consists solely of calling the group's validation function.

Group validation occurs when the following events occur:

1. The cursor leaves the group due to a TAB, or a BACKTAB. Normally, the arrow keys do not cause validation. The library function `sm_option` can be used to cause validation to occur whenever the cursor leaves a group. An individual group can also be validated by calling the library function `sm_gval`.
2. The XMIT key is pressed. All groups on the screen are validated. Note that XMIT is the only key that causes groups to be validated. Other keys can perform the same function if specified by a call to the library function `sm_keyoption`. All of the groups on a screen can be validated by a call to the library function `sm_s_val`.

7.9
VIEWPORTS AND POSITIONING

7.9.1
Introduction

The viewport facility enables the use of *virtual screens* that are larger than their display size. You may even create virtual screens that are larger than the physical display. The viewport facility determines which portion of a virtual screen is visible at a given time, as well as the size and position of the "viewport" into the virtual screen, that appears on the display.

A viewport is always smaller than or equal in size to the virtual screen being viewed, and is never larger than the physical display. If the virtual screen has a border, then scroll bars, each consisting of a pair of arrows, will appear at the bottom and right hand side of the viewport, to indicate that only part of the screen is visible. The size of the scroll bar relative to the size of the border indicates the percentage of the virtual screen that is shown. The position of the scroll bar arrows along the border indicates which section of the screen is visible.

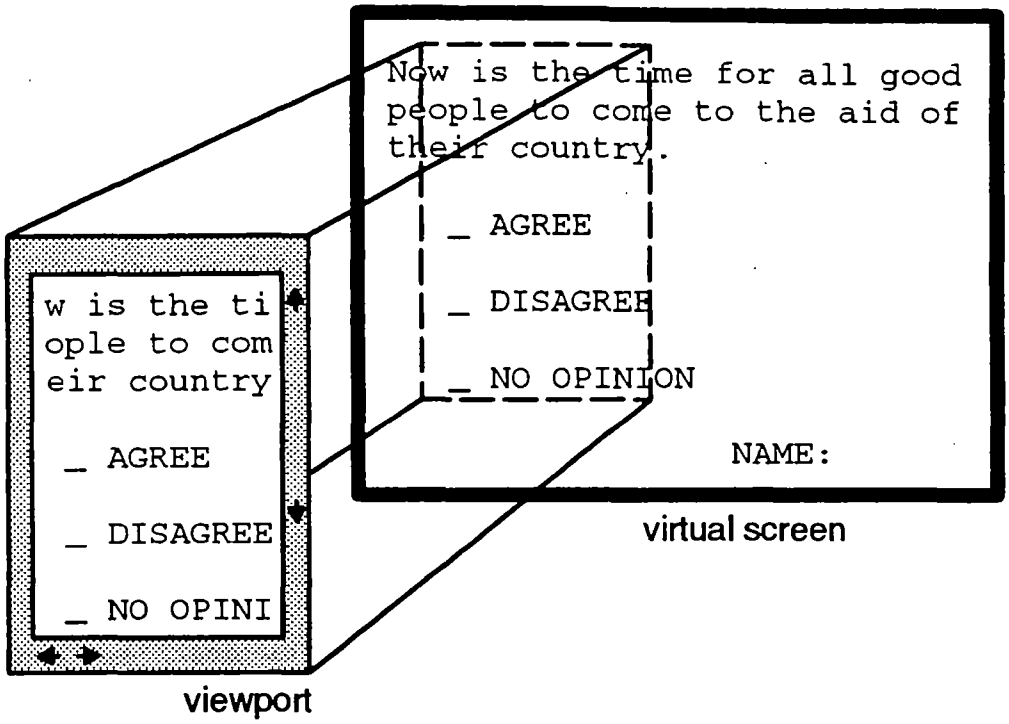


Figure 97: Schematic of a viewport into a virtual screen. Note how the scroll arrows indicate both how much of the virtual screen and which section of it is visible. The position of the horizontal scroll arrows to the left indicates that the viewport is showing the left-hand portion of the virtual screen. Their closeness indicates that the viewport is showing a small horizontal section of the screen. The position of the vertical scroll arrows near the top of the border indicates that the viewport is showing the top portion of the screen. The separation of the arrows indicates that a large vertical section of the screen is shown.

As a user tabs through the fields in a virtual screen, **JAM** automatically scrolls the viewport, bringing the necessary fields into view at the appropriate time. The user may optionally use the VIEWPORT key to move, resize, and scroll the viewport manually. This key also controls sibling windows, discussed on page 126.

7.9.2

Specifying a Viewport

The size of a viewport is specified via the positioning parameters in the control string that displays a screen. It is also specified in the strings that display screen and field help screens, and item selection screens. Please note that when we refer to the size of the physical display, we really mean the portion of the physical display that is available for the presentation of screens (with borders). Typically, one row is reserved for the status line and one or two rows may be reserved for soft key labels. The format specification is as follows:

(***row***, ***col***, ***height***, ***width***, ***vrow***, ***vcol***) ***screen-name***

where:

- ***row*** specifies the row of the physical display at which to position the upper left corner of the viewport. Coordinates are based at 1. A row specification may be in absolute or relative coordinates. Relative coordinates refer to the upper left hand corner of the calling screen. For example, –5 indicates: "display this viewport 5 rows above the top left hand corner of the previous screen".
- ***col*** specifies the column of the physical display at which to position the upper left corner of the viewport. Coordinates are based at 1. Column specifications may also use relative coordinates. A negative number indicates positioning to the left of the underlying screen.
- ***height*** specifies the height of the viewport in rows. A height of zero indicates that the viewport should be as tall as the virtual screen (but never larger than the physical display).
- ***width*** specifies the width of the viewport in columns. A width of zero indicates that the viewport should be as wide as the virtual screen (but never larger than the physical display.
- ***vrow*** specifies which row of the virtual screen to display at the top of the viewport. The border is counted as part of the screen.
- ***vcol*** specifies which column of the virtual screen to display at the left edge of the viewport.

If you specify a ***vrow*** or ***vcol***, then the cursor will appear in the upper left corner of the viewport, whether there is a field there or not. The cursor will respond to the cursor keys freely, until it encounters an unprotected field, or the TAB key is pressed. Once it is in a field, the cursor will behave normally (that is, it will be restricted to moving among unprotected fields).

The positioning parameters are optional. If they are specified, then they may be integer constants, names of fields on the screen, or names of LDB entries. If none are specified, then **JAM** will attempt to display the entire screen in a viewport the size of the screen at such a position that it does not hide the field that the cursor was in when the screen was called (or in the upper left hand corner of the display if the screen is called as a form). If the virtual screen is larger than the display and no parameters are specified, **JAM** will create a viewport the size of the display that contains the first unprotected field (or the upper left corner of the screen if there are no such fields). If only some of the parameters are specified, then **JAM** will use defaults for the unspecified items.

Relative positioning is a powerful tool, as the position of a window becomes context sensitive. If the underlying screen is moved, a relatively positioned window will still appear in the appropriate location when it is called. If necessary, JAM will scroll the calling screen in its viewport in order to display the window at the relative location specified.

Normally, it is not necessary to specify positioning parameters, but if you do, keep in mind that in order to use, for example, the ***vcol*** parameter, you must specify all of the parameters to the left of it.

Below are some sample viewport specifications:

- Display the screen `smoo` as a window at row 6, column 7 of the physical display:

```
&(6,7)smoo
```

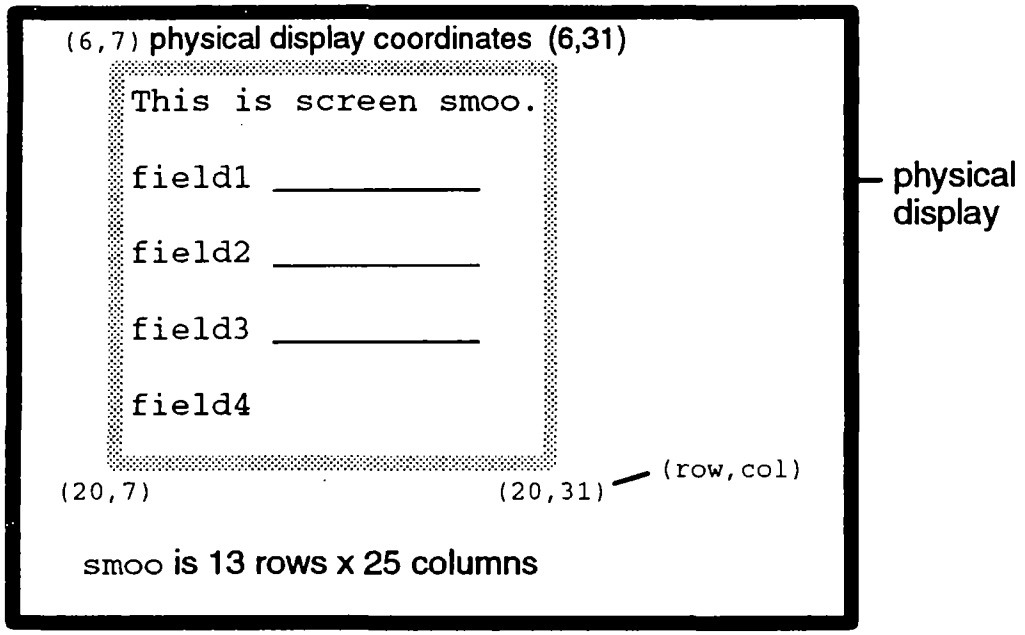


Figure 98: Viewport Specification: `&(6,7)smoo`

- Display smoo as a window at row 6, column 7, in a viewport 8 rows tall and as wide as the screen. The first unprotected field in smoo will appear within the viewport:

```
&(6,7,8,0)smoo
```

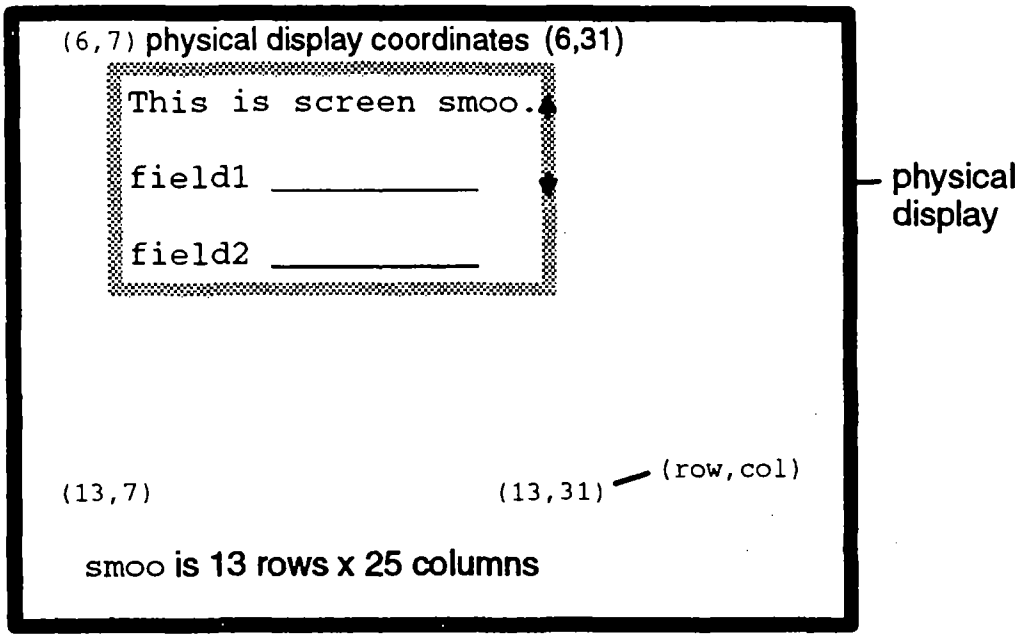


Figure 99: Viewport Specification: &(6,7,8,0)smoo

- Display smoo as a sibling window at row 6, column 7, in a viewport 8 rows tall and 9 rows wide, with row 4, column 5 of smoo at the top left.

```
&&(6,7,8,9,4,5)smoo
```

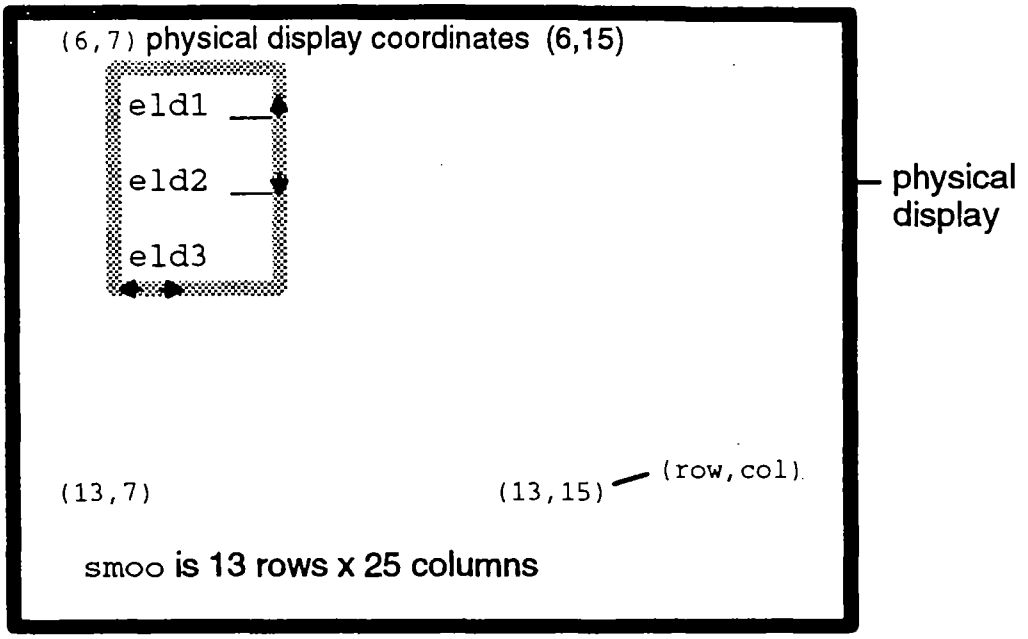


Figure 100: Viewport Specification: `&(6,7,8,9,4,5)smoo`

- Display `smoo` as a stacked window 7 rows below and 6 rows to the left of the calling screen, in a viewport 5 rows tall and as wide as the screen.

`&(+7,-6,5,0)smoo`

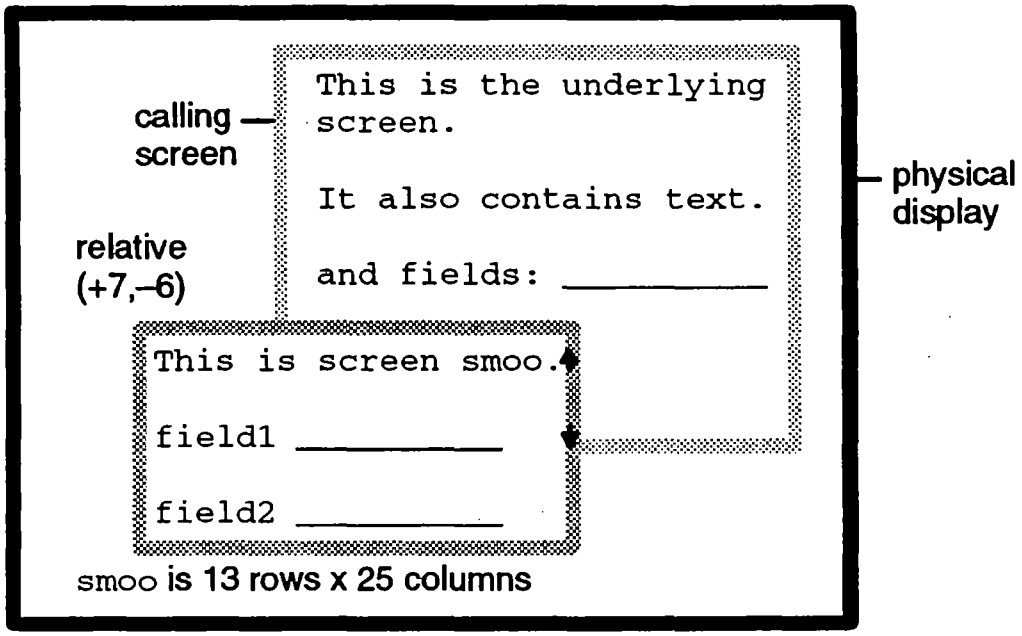


Figure 101: Viewport Specification: `&(+7,-6,5,0)smoo`

7.9.3

The VIEWPORT Key

The developer may assign a key to the logical viewport function. This key allows the user to move, resize and scroll (offset) the viewport. It also may be used for selecting sibling windows. In application mode of `jxform`, SPF9 (labelled `window` on the status line) performs the same function as the VIEWPORT key. Pressing the VIEWPORT key brings **JAM** into viewport mode. Asterisks will appear at the corners of the active viewport. The following functions are enabled in viewport mode and are displayed on the status line (note that the selected function will appear in all caps on the status line):

- `PF2move`

  This is the default when you enter viewport mode. If the viewport is smaller than the physical display, select this function to reposition the viewport. Press PF2 and use the cursor keys to position the asterisks at the desired location. Note that only the asterisks will move. Press PF3, PF4, or PF5 to anchor the viewport and begin another viewport function; or press XMIT or EXIT to anchor the viewport and leave viewport mode.

- `PF3resize`

  Press PF3 and use the cursor keys to resize the viewport from the lower right hand corner. When the asterisks indicate the desired size, press another function key to confirm. At this point, the screen will be updated. When the viewport is the size of the virtual screen, it can no longer be expanded.

- `PF4offset`

  Press PF4 and use the cursor keys to scroll (change the offset of) the virtual screen within the viewport. Press XMIT (or another function key) when finished. **JAM** automatically scrolls the viewport as you tab through fields in order to bring the next field into view.

- `PF5next window`

  Press PF5 to activate the next open sibling window. The asterisks will move to the selected window. The PF5 key may be pressed repeatedly to cycle through all available sibling windows. Press XMIT (or another function key) to confirm your selection. The creation of sibling windows is discussed on page 126.

Press EXIT or XMIT to leave viewport mode. The selected sibling window will be active when you leave viewport mode.

If the mouse is supported for **JAM** on your platform, it may be used to perform viewport key functions. Refer to the mouse document.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

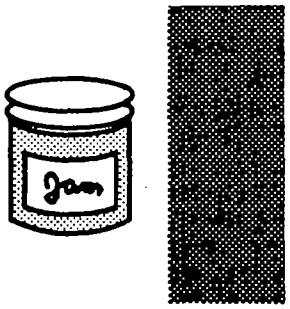
X

Y

Z

Configuration Guide

TABLE OF CONTENTS



Chapter 1
Introduction

The flexibility of **JAM** is twofold. It may be adapted for both hardware and software needs. This part of the manual, the **JAM** *Configuration Guide*, describes how you can make these changes.

The **JAM** configuration directory (`config`) includes the following ASCII and binary files:

- `vid, vid.bin`
- `keys, keys.bin`
- `msgfile, msgfile.bin`
- `smvars, smvars.bin`

By using these files you can adapt **JAM** to the following:

- the terminals attached to your system
- the layout of your system
- the defaults for **JAM**'s behavior
- the content and style of messages

When configuring **JAM** for the terminals attached to your system, you must have a video and a key translation file for each type of terminal. The video translation file tells **JAM** how to drive the display, and the key translation file tells **JAM** how to interpret the character sequences produced by the keyboard attached to a terminal. **JAM** is compatible with any ANSI terminal, and it has over fifty key and video files for specific vendor terminals. In the unlikely event that none of the distributed key and video files work with your terminal, these files may also be built from scratch with help from this guide. However, most readers will probably use these two chapters for reference and for customizing distributed video and key files.

When configuring **JAM** to the layout of your system, you must tell **JAM** where to find its screen and help libraries, its configuration files, etc. Typically, these paths are specified in an `smvars` file. An environment variable `SMVARS` is set to the directory path to the binary version of this file.

When setting defaults for **JAM** behavior, you may use the `smvars` file to specify the color of error windows, the methods of acknowledging error messages, and the behavior of cursor control keys like Tab or arrow keys, etc.

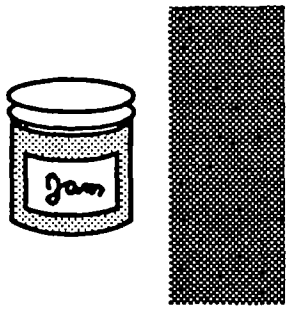
Finally, when adapting the content and style of messages for developers and users, the message file may be modified. All text in **JAM** comes from the message file. "Please hit the space bar" may be changed to "Press the space bar to resume application". This file also contains the default values for `SM_YES`, `SM_NO`, days of the week, months of the year, as well as date/time and currency formats.

The remainder of this guide is divided into four chapters—one for each of these files in the **JAM** `config` directory[1]. As we discuss each file, we will also suggest changes and enhancements you may wish to make to the file. You may need to modify none of them, or you may need to modify all of them. If you make changes, you will be modifying the ASCII version of a configuration file. Since **JAM** always uses the binary versions of these files at initialization, you must convert a file to binary after editing it. Each file has its own utility for this conversion—`vid2bin` for video files; `key2bin` for key files; `msg2bin` for message files; `var2bin` for setup files. The **JAM** *Utilities Guide* has more information on these utilities.

1.1
CONVENTIONS USED

- `literal` We use this font for words which you will type verbatim. In particular, we use this font for all our examples. In addition, when we name a configuration file, a configuration keyword, **JAM** utility, or **JAM** library function we use this font to distinguish it from the standard text.
- ***italics*** We use bold italic to show where file names, configuration values, and other values should appear. You should replace these with the appropriate names in your applications.

1. In most Release 5 installations, **JAM** screen and help libraries are also stored in the `config` directory. These libraries are usually not altered. However, to permit the localization of **JAM** for non–English speaking developers, these libraries may be adapted after JYACC has granted the appropriated licenses. The conversion or modification of these libraries is not documented in this guide.



Chapter 2
Key Translation File

2.1
INTRODUCTION

To build a **JAM** application, you must be able to enter ASCII data characters (i.e., `A`, `q`, `8`, `!`, `7`, `?`, `]`, ...) and to indicate certain logical key values to **JAM**—`EXIT`, `XMIT`, `PF1`, `SPF1`, etc. Since physical keyboards vary from system to system, **JAM** uses a key translation file to translate a sequence typed to a physical keyboard into a logical key which **JAM** understands. The logical keys are defined as hexadecimal values in the include file `smkeys.h`. This file is terminal–independent, while a key translation file is terminal–dependent. `ibmkeys`, `vt220keys`, `wy75keys` are a few of the available key translation files. **JAM** provides a `modkey` utility to help you build and edit key translation files.

Regardless of your system, we expect all keyboards to have keys for the ASCII data characters; these data keys require no translation and are not included in the key translation file. We cannot expect different keyboard types, however, to have the same number of function keys, or to have logical keys named HELP and TRANSMIT. Therefore, a key translation file must specify a physical key which will work as a HELP key, and another as a TRANSMIT, and so on. When you press that key during a **JAM** application **JAM** looks at the key translation file and matches the key sequence with a logical value. In this way, **JAM** is able to interpret TRANSMIT when you press a "Do" key, or an "End" key, or whatever physical key is mapped to TRANSMIT in your key translation file.

Pressing a physical key transmits a unique code or sequence of codes; in a key translation file, this key never has more than one logical value. For example, "Return" key may have the logical value TRANSMIT, or it may have the logical value NL (new line),

but it cannot have both values in the same key file. If a timing interval is specified in the video file, then one key may use another key's sequence as a lead–in. For example, if the F1 key on a particular keyboard sends `ESC [ a`, you can still use `ESC` itself as a **JAM** key. If no timing interval is specified, then one key cannot contain another key as a lead–in. Refer to section 5.4.1 on page 77 for a instructions on setting an interval via `KBD_DELAY`.

Not surprisingly, with a one–to–one mapping there are not enough keys on a commercially available physical keyboard to represent the **JAM** logical keyboard. To accommodate the larger logical keyboard, we combine two or more physical keys to represent a logical key. For example, on a PC we often map the logical key ZOOM to Alt–Z.

If you use **JAM** on terminals with different operating systems and keyboards then each terminal must have its own key translation file. If necessary, you may also create more than one key translation file for a terminal. This way you can tailor the key mapping for a particular application. Another, and usually better, solution is to create application keysets.

A keyset is a "soft" keys translation file. A keyset allows you to use the same small set of keys to represent different **JAM** keys at different times, in an application–dependent manner. Normally, a keyset uses eight function keys. The terminal's key translation file tells **JAM** what sequence a soft key actually sends. This sequence is then translated based on the active keyset. An application may use any number of keysets. For example, every screen in an application could have its own keyset. When the screen is opened, the labels defined in its keyset would appear in a row on the screen. In addition, a keyset may specify more than one row, so that when a key is pressed, the next row of the keyset is displayed on the screen. A screen without a specified keyset uses a default keyset; if the screen is brought up as a window it will use the keyset of the form beneath it.

JAM provides a keyset editor for defining keysets. You must modify `jxmain.c` and `jmain.c` to create and use keysets. See the **JAM** *Programmer's Guide* for details on modifying these files. See the **JAM** *Author's Guide* for instructions on using the keyset editor. To display key labels, you must modify the video file. See the video file chapter in this guide.

In summary, `smkeys.h` defines the independent **JAM** logical keyboard. A key translation file, like `ibmkeys` or `wy85keys`, maps the ASCII values returned from a physical keyboard to the names of logical keys; it is terminal–dependent. A keyset maps a soft key to the name of a logical key; it is designed for a specific application. Keysets are terminal independent, except that the number of keys per row may be tailored for a particular terminal.

The rest of this chapter describes:

- How to read the entries in the key translation file (page 5);
- How to modify a key translation file (page 11);
- Using alternate key translation files in portable applications (page 13).

Please see the chapter on keyboard input in the *Programmer's Guide* for a discussion of key processing in **JAM** applications.

2.2
KEY TRANSLATION FILE SYNTAX

Each entry in a key file has the form:

logical-key*(*key-label*) = *character-sequence

logical-key is the mnemonic or the hexadecimal value of a **JAM** logical key. For example, the logical TRANSMIT key is represented by the mnemonic `XMIT` or by the hex value `0x104`. The mnemonics and hex values for all the **JAM** logical keys are defined in `smkeys.h`. Beginning on page 6 is a list of these mnemonics and values.

Whether you use the mnemonic or the hexadecimal value, there will be no difference when executing the application or when using `modkey`. For example, whether you use `XMIT` or `0x104` for ***logical-key***, its definition will be displayed in the `modkey` window "Define Cursor Control and Editing Keys." See the *Utilities Guide* for more information.

You may assign the same ***logical-key*** two (or more) different ***character-sequences***. For example, in a key file for the PC you might make these entries:

```
HELP(Ctrl F1) = NUL ^
HELP(Alt H) = NUL #
```

When this key file is used with a PC, both Ctrl–F1 and Alt–H will execute `HELP`. In `modkey`, the first entry will be displayed in the window for "Define Cursor Control and Editing Keys" and the second entry for `HELP` will be displayed in "Define Miscellaneous Keys."

key-label, which must be enclosed in parentheses, is the letter, numeral, character, or character string engraved on a physical keytop. One or more physical key labels may be included inside the parentheses, for example (`Alt F1`). ***key-label*** is optional, but very useful. It is stored in the key translation file and can be accessed at run–time through various library functions and the `%K` escape in status–line messages. (See `d_msg_line` in the **JAM** *Programmer's Guide*.) Key labels are often helpful in user messages and prompts.

character–sequence follows the equal sign. It is up to six characters long, not including blanks. **JAM** will translate ***character–sequence*** to be the logical key on the left of the equals sign.

When a physical key is pressed, it transmits a character code which is unique from the code produced by any other key on the keyboard. ***character–sequence*** is the sequence of characters produced by a keystroke. Each complete character sequence may have only one logical value. Although one sequence may include another as a substring. If you assign the same character–sequence more than once in `modkey`, **JAM** will display an error message. If you are using a text editor to create a key translation file, **JAM** displays an error message when you attempt to convert the file with `key2bin`. If you assign a character sequence that includes another key's sequence as its lead–in, `modkey` will issue a warning that states "Key overlaps with a FUNCTION key.", but it will allow the sequence.

Lines beginning with a pound sign # are treated as comments. They are ignored by the conversion utility `key2bin`.

Below are some sample entries from a key translation file.

```
EXIT (F1) = SOH @ CR
XMIT (Enter) = SOH O CR
TAB = HT
BACK = NUL SI
BKSP = BS

# These are the arrow keys
RARR = ESC [ C
LARR = ESC [ D
UARR = ESC [ A
DARR = ESC [ B

# The next entry uses a hex value rather
# than a mnemonic for logical-key
0x108 = DEL
```

2.2.1

Key Mnemonics and Values

The following list is taken from the include file `smkeys.h` which defines the **JAM** logical keyboard. We marked the entries required by `jxform` with "**" and the recommended entries with "*".

Logical Key Mnemonic	*Hex Value*	*Description*
EXIT	0x103	exit**
XMIT	0x104	transmit**
HELP	0x105	help on field*
FHLP	0x106	help on screen or form
BKSP	0x108	backspace*
TAB	0x109	tab*
NL	0x10a	newline*
BACK	0x10b	backtab*
HOME	0x10c	go to first field on screen*
DELE	0x10e	delete character*
INS	0x10f	insert/overwrite character toggle*
LP	0x110	local print
FERA	0x111	field erase*
CLR	0x112	clear all unprotected*
SPGU	0x113	scroll up a page
SPGD	0x114	scroll down a page
LSHF	0x116	left shift
RSHF	0x117	right shift
LARR	0x118	left arrow*
RARR	0x119	right arrow*
DARR	0x11a	down arrow*
UARR	0x11b	up arrow*

Logical Key Mnemonic	*Hex Value*	*Description*
REFR	0x11e	refresh screen*
EMOH	0x11f	go to last field on screen
INSL	0x120	insert line*
DELL	0x121	delete line*
ZOOM	0x122	zoom on field*
SFTS	0x123	soft key select
MTGL	0x124	toggle menu mode
VWPT	0x125	adjust viewport
MOUS	0x126	indicate mouse event
SFTN	0x1002	select next set of soft keys
SFTP	0x1003	select previous set of soft keys

Note: The documentation on error messages and error acknowledgment often refers to an "error acknowledgment key" whose default is the space bar. Since the space bar is a data entry key, it cannot used as a logical key. Instead, the key is defined as the setup variable ER_ACK_KEY. It may changed in the `smvars` file, in a setup file, in the system environment, or at runtime with the library function `sm_option`. See the chapter on setup files in this guide for more information.

In the table below, we list the mnemonics and hexadecimal values for function keys, shifted function keys, application function keys, and soft keys.

PF	*Hex*	*SPF*	*Hex*	*APP*	*Hex*	*SFT*	*Hex*
PF1	0x6101	SFP1*	0x4101	APP1	0x6102	SFT1	0x6105
PF2*	0x6201	SPF2*	0x4201	APP2	0x6202	SFT2	0x6205
PF3*	0x6301	SPF3*	0x4301	APP3	0x6302	SFT3	0x6305
PF4*	0x6401	SPF4*	0x4401	APP4	0x6402	SFT4	0x6405
PF5*	0x6501	SPF5*	0x4501	APP5	0x6502	SFT5	0x6505
PF6*	0x6601	SPF6*	0x4601	APP6	0x6602	SFT6	0x6605
PF7*	0x6701	SPF7	0x4701	APP7	0x6702	SFT7	0x6705
PF8*	0x6801	SPF8	0x4801	APP8	0x6802	SFT8	0x6805
PF9*	0x6901	SPF9	0x4901	APP9	0x6902	SFT9	0x6905
PF10*	0x6a01	SPF10	0x4a01	APP10	0x6a02	SFT10	0x6a05
PF11	0x6b01	SPF11	0x4b01	APP11	0x6b02	SFT11	0x6b05
PF12	0x6c01	SPF12	0x4c01	APP12	0x6c02	SFT12	0x6c05
PF13	0x6d01	SPF13	0x4d01	APP13	0x6d02	SFT13	0x6d05
PF14	0x6e01	SPF14	0x4e01	APP14	0x6e02	SFT14	0x6e05
PF15	0x6f01	SPF15	0x4f01	APP15	0x6f02	SFT15	0x6f05
PF16	0x7001	SPF16	0x5001	APP16	0x7002	SFT16	0x7005
PF17	0x7101	SPF17	0x5101	APP17	0x7102	SFT17	0x7105
PF18	0x7201	SPF18	0x5201	APP18	0x7202	SFT18	0x7205
PF19	0x7301	SPF19	0x5301	APP19	0x7302	SFT19	0x7305
PF20	0x7401	SPF20	0x5401	APP20	0x7402	SFT20	0x7405
PF21	0x7501	SPF21	0x5501	APP21	0x7502	SFT21	0x7505
PF22	0x7601	SPF22	0x5601	APP22	0x7602	SFT22	0x7605
PF23	0x7701	SPF23	0x5701	APP23	0x7702	SFT23	0x7705
PF24	0x7801	SPF24	0x5801	APP24	0x7802	SFT24	0x7805

2.2.2

ASCII Character Mnemonics and Hex Values

This table lists two– and three–letter ASCII mnemonics for control and extended control characters:

Mnemonic	*Hex*	*Mnemonic*	*Hex*	*Mnemonic*	*Hex*	*Mnemonic*	*Hex*
NUL	0x00	DLE	0x10			DCS	0x90
SOH	0x01	DC1	0x11			PU1	0x91
STX	0x02	DC2	0x12			PU2	0x92
ETX	0x03	DC3	0x13			STS	0x93
EOT	0x04	DC4	0x14	IND	0x84	CCH	0x94
ENQ	0x05	NAK	0x15	NEL	0x85	MW	0x95
ACK	0x06	SYN	0x16	SSA	0x86	SPA	0x96
BEL	0x07	ETB	0x17	ESA	0x87	EPA	0x97
BS	0x08	CAN	0x18	HTS	0x88		
HT	0x09	EM	0x19	HTJ	0x89		
NL	0x0a	SUB	0x1a	VTS	0x8a		
VT	0x0b	ESC	0x1b	PLD	0x8b	CSI	0x9b
FF	0x0c	FS	0x1c	PLU	0x8c	ST	0x9c
CR	0x0d	GS	0x1d	RI	0x8d	OCS	0x9d
SO	0x0e	RS	0x1e	SS2	0x8e	PM	0x9e
SI	0x0f	US	0x1f	SS3	0x8f	APC	0x9f

SP	0x20	DEL	0x7f

2.3
MODIFYING KEY TRANSLATION FILES

Modifying a key translation file involves 3 steps:

1. Use a text editor or the `modkey` utility to create a new ASCII key file. You can modify one of the ASCII key files supplied with **JAM**, or create a completely new key file.
2. Use `key2bin` to create a binary version of the ASCII key file.
3. Change the value of the `SMKEY` setup variable to the name of the new binary key file. During initialization, **JAM** reads the key file specified by `SMKEY` into memory. (The default value for `SMKEY` is set in the `config` file `smvars`. If you modify `smvars`, you must convert it to binary with the utility `var2bin`. See the chapter on setup files in this guide for more information.) The choice of `SMKEY` is terminal dependent.

2.3.1
Changing Keys for Users and Developers

You may wish to change the mappings of logical keys for the preferences of users or developers. For example, in the distributed key files for the PC, `XMIT` is mapped to the physical "End" key, and `NL` is mapped to the "Enter" key. The key translation file entries are:

```
XMIT(End) = NUL O
NL(Enter) = CR
```

`NUL O` is the sequence transmitted by the PC's "End" key; when **JAM** receives this sequence from the keyboard it carries out its `XMIT` function. Similarly, `CR` is transmitted by the PC's "Enter" key and **JAM** responds appropriately.

If you prefer to use the "Enter" key for `XMIT` and the "End" key for `NL`, you could change the key file entries to read:

```
XMIT(End) = CR
NL(Enter) = NUL O
```

The Enter key would be used for `TRANSMIT`, and the End key would be used where newline is needed, like word-wrapped arrays. Remember that changes made to the key file will affect the entire application. You may use the library function `sm_keyoption` to change the behavior of logical keys at run-time. See your *Programmer's Guide* for more information.

Accessing Extended Keys on the PC

Developers who are building applications for PCs with extended keyboards may wish to use the additional keys and key combinations provided. By default, these keys are not recognized in the distributed PC version of **JAM**. Before these keys can be defined in `modkey`, you must add a special flag for each class of key or key combination to the video file entry `INIT`. The classes are listed in the table below.

Flag	*Description*
`XKEY`	Allows access to the `F11` and `F12` function keys.
`GRAYKEYS`	Distinguishes between gray and white cursor positioning keys. For example, this option allows you to assign one value to the gray up arrow key and another value to the white up arrow key.
`MULTISHIFT`	Permits the use of sequences using the key combinations: `Ctrl-Alt-`, `Shift-Alt-`, and `Ctrl-Shift-Alt-`

For example, an `INIT` sequence that activates the `F11` and `F12` keys might look like,

```
INIT = C 0, 7, 2, XKEY
```

After converting the altered video files with `vid2bin`, you may then define the keys with `modkey`. For example, you might define `F11` and `F12` in the "Define Miscellaneous Keys" window as supplementary HELP and FHLP keys. Their definitions in the ASCII file would appear as

```
HELP (F11) = NUL NEL
FHLP (F12) = NUL SSA
```

Extended keys are documented in the PC video files. The `INIT` entry is fully documented in the Video File chapter of this guide.

2.3.2

Using International and Composed Characters

JAM accepts and interprets 8–bit international characters automatically. Some terminals, however, use character sequences which correspond to international characters. Some terminals also allow the user to compose characters (by programming a key to transmit a specified character sequence). To support either situation, you first should assign the sequence to a hex value in the range `0xA0 - 0xFE` in the key file. A corre-

sponding entry in the GRAPH table of the video file would specify the display (see Graphics and Foreign Character Support in the Video File chapter in this guide). Without a GRAPH entry, the 8–bit character is transmitted as is, and must be interpreted by the terminal.

2.4
USING ALTERNATE KEY TRANSLATION FILES

Many applications must support more than one type of keyboard. **JAM** allows you to provide this support without recompiling the application for each keyboard. Each terminal must have a working key file and video file. The application's `smvars` file should list the paths for the available key and video files. If the user's SMTERM variable is set correctly, **JAM** will select the correct key and video file from the `smvars` file during initialization. Please see the Setup File chapter in this guide for more information.

Chapter 3
Message File

3.1
INTRODUCTION

The **JAM** libraries use the configuration file `msgfile` as the text source of messages and date strings. The text in `msgfile` may be adapted for developers' or end users' needs. Since **JAM** uses binary configuration files at runtime, you must convert any new or modified message files with the utility `msg2bin`.

JAM uses the include file `smerror.h` to define identifiers (manifest constants) for all the messages used by the **JAM** libraries. In `msgfile`, these identifiers are "system tags" which are assigned message text. If user messages are added to the file, **JAM** numbers them consecutively starting at zero, when the file is converted with `msg2bin`. During initialization, **JAM** looks for the environment variable `SMMSGS`. This variable gives the full pathname of a binary message file; by default, it is `msgfile.bin`.

There are obvious advantages to storing message text in binary files. Messages do not need to be "hard coded" for either **JAM** or a **JAM** application. When developing an application, message text is edited and compiled without recompilation of the application. Storing **JAM**'s system messages in an accessible binary file means that any message may be modified to suit the needs of developers. In fact, the message text in the message file may be translated to another language. If you created a French version of the message file, the status line messages which identify physical keys with logical values would all appear in French. System dates would use French names for the days of the week and months of the year. Formats for date and currency edits would be adapted to French standards. It might be useful to translate some error messages as well; for example, "Entry is required." However, the translation of run–time error messages is complicated since we do not document when and what functions call a particular error message.

Although you may not need to modify any of the entries in the message file, you will probably want to add messages and create application message files. This chapter explains how to do this. In addition, many **JAM** defaults are defined in the message file (the date and currency formats; the values of `SM_YES`, `SM_NO`, day and month mnemonics). Familiarity with the message file may help you with functions that use these defaults.

The rest of this chapter describes:

- Reading entries in the message file (page 16);
- Modifying a message (page 17);
- Adding message entries and creating message files (page 18);
- Embedding attributes and key names in messages (page 19);
- Customizing date and time formats (page 22);
- Customizing currency formats (page 29);
- Using alternate message files (page 31).

3.2
MESSAGE FILE SYNTAX

Each entry in a message file has the form:

TAG = ***Message***

TAG is a single word (no embedded blanks). It may include letters, digits, and underscores.

If ***TAG*** identifies a system message, it is defined in the `include` file `smerror.h` and it begins with a standard prefix. Below is a list of the system prefixes; these prefixes are reserved and may not be used for other messages.

- `SM` messages and strings used by the **JAM** run–time library.
- `FM` messages issued by the Screen Editor.
- `JM` additional run–time messages used by **JAM**.
- `JX` messages issued by `jxform`.
- `UT` messages issued by **JAM** utilities.

The equals (=) sign following ***TAG*** is required. Blanks are allowed both before and after the equals sign.

Message can be any alphanumeric string that does not contain carriage returns. If ***Message*** is longer than one line, use a backslash to end the lines which are continued. For example,

```
PQ_FATALERR = Application unable to post your \
transaction. Contact your system manager.
```

JAM will automatically display long status–line messages in a window, so that the entire message is visible. ***Message*** may also contain percent sequences that specify appearance, positioning, and acknowledgement information. (See page 19.)

You may include comments in the message file by beginning the comment line with a pound sign (#). These lines will be ignored when the file is compiled with the utility `msg2bin`.

If ***Message*** begins and ends with the same quote character, **JAM** will strip off the quotes when displaying the message. Use quote characters, then, to display leading blanks in a message.

An excerpt from a sample message file is shown below.

```
SM_RENTRY = Entry is required.
SM_MUSTFILL = Must fill field.
SM_CKDIGIT = Check digit error.
SM_NOHELP = No help text available.

# The following are user messages.
US_INSUF = Insufficient funds.
RESERVED =
US_SUPV = See supervisor.
```

When the file containing these messages is compiled with `msg2bin`, the utility uses the ***TAG***s to distinguish between system and user messages. It assigns the user messages consecutive numbers starting from zero; the definitions of system message are taken from `smerror.h`. It is the responsibility of the application programmer to maintain the ordering of user messages and the assignment of their identifiers. See Section 3.4 for more information.

While a prefix is not required in a user ***TAG***, it is often useful. All entries with the same prefix may be loaded into memory with a call to the library function `sm_msgread`.

3.3
MODIFYING MESSAGES

The ASCII version of the message file can be modified using a text editor. After making the changes, run the utility `msg2bin` to convert the file to binary.

For example, if the file was modified as follows:

```
SM_DAYL1 = domingo
SM_DAYL2 = lunes
SM_DAYL3 = martes
SM_DAYL1 = miércoles
SM_DAYL1 = jueves
SM_DAYL1 = viernes
SM_DAYL7 = sábado
```

the long names for days of the week would be displayed in Spanish, rather than English, in date edits using the system clock.

3.4
ADDING MESSAGES

New messages may be added to the message file. As described in the previous section, the new message must have a unique ***TAG*** which does not begin with any of the system prefixes. `msg2bin` will number the new messages consecutively starting from zero. To use the message, you must define its ***TAG*** in the application program according to the assignment made by `msg2bin`.

For example, if you added these three entries to the message file,

```
US_INSUF = Insufficient funds.
RESERVED =
US_SUPV = See supervisor.
```

then an application program compiled with the following definitions:

```
#define US_INSUF   0
#define RESERVED   1
#define US_SUPV    2
```

could issue these calls:

```
sm_quiet_err (sm_msg_get (US_INSUF));
sm_err_reset (sm_msg_get (U_SUPV));
```

If you needed to change the text of a new message, you would modify the message file and convert it to binary with `msg2bin`. You would not need to recompile the application program.

If any message is missing from the message file and a call is made to display the message, only the message number (from the `#define` statement) is displayed. For example, if the entry for `SM_RENTRY` was deleted from the message file, and an end–user failed to enter data in a field where entry was required, the status line would display the number corresponding to `SM_RENTRY` in `smerror.h`.

User messages may also be stored in separate message files, loaded with calls to `sm_msgread`, and accessed in the same way as described above.

3.5
EMBEDDING ATTRIBUTES AND KEY NAMES IN MESSAGES

Several percent escapes provide control over the content and presentation of status messages. The character(s) following the percent sign are case–sensitive and must be typed in exactly as described. This will avoid conflicts with percent escapes used by `printf` and the tokens used by date/time formats. Some percent escapes must appear at the beginning of the message, or may be used only by error messages; the restrictions are noted in the appropriate cases. The percent escapes are the following:

For all messages:

- `%A` Change display attributes.
- `%K` Display key label.
- `%B` Beep the terminal.

For error messages requiring acknowledgement:

- `%N` Use a carriage return in the message text and display the message in pop–up window.
- `%W` Display message in a pop–up window.
- `%Md` Force the user to acknowledge an error message.
- `%Mu` Permit any keypress to serve as both error acknowledgement and data entry.

3.5.1
%A – Change Display Attributes

%A*hhhh* placed anywhere in the text of a message changes the display attributes of the text that follows it. ***hhhh*** is a hexadecimal number which will be interpreted as a display attribute. It may represent one attribute or the sum to two or more attributes. If you wish to display a digit immediately after the attribute change, pad the attribute to four digits with leading zeros. If the character following the attribute change is not a legal hex digit, the leading zeros are unnecessary. The listing below is taken from the definitions given in the `include` file `smattrib.h`.

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Attributes		Background Attributes	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		
Foreground Colors		Background Colors	
BLACK (colors are additive)	0000	B_BLACK (default)	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700
NORMAL_ATTR (usually white)	0007		

For example:

```
SM_WARNBIG= %A44Warning.\
   %A0004Form is larger than screen size.
```

This would cause the message to appear in red characters against the default black background with "Warning." in blinking characters. You can use %A escape sequences with all status–line and error messages. It overrides any setup variable assignment which may have altered the defaults for message attributes.

If you use %A without specifying a color, the default foreground color is BLACK. Since the default background for the status line is also black, developers using color terminals should include a color code when using %A.

3.5.2
%K – Display Key Label

%K***logical–key*** placed anywhere in the text of a message causes **JAM** to interpret ***logical–key*** as a mnemonic defined in smkeys.h. If a key label has been defined for the logical key in the key translation file, the key label will replace the percent sequence in the message text. If there is no key label (or no such logical key) %K is stripped off and ***logical–key*** remains in the message text. For example, if the key translation file contains the entry:

```
XMIT(End) = NULL O
```

and the message file contains the entry:

```
SG_CONFIRM = Press %KXMIT to confirm.
```

the status line would display:

```
Press End to confirm.
```

You can use a %K sequence in all status–line and error messages.

Note: If %K is used on the status line, and a user clicks on the keylabel text with a mouse, **JAM** responds as if the user had pressed the physical key. See Mouse Support for details on mouse installation and use.[2]

3.5.3
%B – Beep the Terminal

%B in a status–line or error message rings the terminal bell (using sm_bel) when the message is displayed. You may configure **JAM** to send a "visible" bell, such as flashing the screen, by putting a BELL entry in the video file. See the section on **Indicators** in the Video File Chapter.

3.5.4
%N – Use a Carriage Return in Message Text

%N anywhere in an error message is replaced with a carriage return. Any message with one or more %N's is automatically displayed in a pop–up window, usually at the bottom of the screen.

2. Mouse support is available for PC users. Documentation is included with the PC releases of **JAM**.

3.5.5
%W – Display Message in a Pop–up Window

%W at the beginning of an error message causes the message to be displayed in a pop–up window rather than on the status line. The window will appear at the bottom center of the screen unless it hides the current field. In that case, the window will appear at the top center of the screen. If you precede %W with %K, %A, or message text, the %W will be treated as part of the message text and not as a percent escape.

3.5.6
%Md – Force User to Acknowledge Error Message

%Md at the beginning of an error message forces the user to press the acknowledgment key to clear the message. This is usually the default behavior for error messages. If you have altered the default by using the library function sm_option or by altering the setup file, you can use this percent sequence to force the user to press the acknowledgement key to clear a message. The keypress will not be processed as data. %Md will not work if it follows a %A, %K, or message text.

The acknowledgment key is usually the space bar. This default may be changed in a setup file with the variable ER_ACK_KEY.

3.5.7
%Mu – Use Any Key to Acknowledge Error Message

%Mu at the beginning of an error message causes the next key press to be both message acknowledgment and keyboard input. %Mu will not work if it follows a %A, %K, or any message text.

3.6
CUSTOMIZING DATE–TIME FORMATS

The formats, substitution text, and substitution variables for displaying dates and time are defined in the message file.

3.6.1

Introduction

You may modify date/time entries to achieve any of the following:

- Customized formats for date/time fields.
- Customized formats and translated text for date/time fields in non–English applications.
- Customized formats, translated text, and translated mnemonics for date/time fields in a non–English version of **JAM**.

In `jxform`, if you specify a date/time format as `DEFAULT`, the field will contain the current date and time in the form

```
11/30/90 13:05
```

This format may be changed so that when `DEFAULT` is specified, the date appears in another format, such as

```
November 30, 1990 - 1:05 PM
```

This is done by changing the tokens assigned to the message entry `SM_0DEF_DTIME`.

If you are customizing the message file for non–English applications, you will need to translate the text entries which name the days of the week and the months of the year. This text is assigned to the entries `SM_DAYA1` ... `SM_DAYA7`, `SM_DAYL1` ... `SM_DAYL7`, `SM_MONA1` ... `SM_MONA12`, `SM_MONL1` ... `SM_MON12`. You may also wish to change the format for `DEFAULT` to reflect local customs; this is done by modifying `SM_0DEF_DTIME`. With these modifications, a date in a field with the format `DEFAULT` could appear as

```
30 novembre 1990  1:05 PM
```

If you are customizing **JAM** for non–English speaking developers, you will translate many of the messages in the message file. In addition to translating the text for the days of the week and months of the year and localizing formats, you may also translate the names of substitution variables. In `jxform`, then, French–speaking developers could use substitution variables like `MOIS2`, `ANNÉÉ4`, `JOURA`, while Spanish–speaking developers could use variables like `MES2`, `AÑO4`, and `DÍAA`, rather than using the English variables `MON2`, `YR4`, and `DAYA`.

3.6.2

The Defaults

When using `jxform` with the distributed message file, these substitution variables are equivalent. These can be changed by the developer:

Shortcut Mnemonic	*Equivalent*
DEFAULT	MON/DATE/YR2 HR:MIN2
DEFAULT DATE	MON/DATE/YR2
DEFAULT TIME	HR:MIN2
DEFAULT3	MON/DATE/YR2 HR:MIN2
DEFAULT4	MON/DATE/YR2 HR:MIN2
DEFAULT5	MON/DATE/YR2 HR:MIN2
DEFAULT6	MON/DATE/YR2 HR:MIN2
DEFAULT7	MON/DATE/YR2 HR:MIN2
DEFAULT8	MON/DATE/YR2 HR:MIN2
DEFAULT9	MON/DATE/YR2 HR:MIN2

Below is an excerpt from `msgfile` which defines the names of the default substitution variables.

```
FM_0MN_DEF_DT  =   DEFAULT
FM_1MN_DEF_DT  =   DEFAULT DATE
FM_2MN_DEF_DT  =   DEFAULT TIME
FM_3MN_DEF_DT  =   DEFAULT3
FM_4MN_DEF_DT  =   DEFAULT4
FM_5MN_DEF_DT  =   DEFAULT5
FM_6MN_DEF_DT  =   DEFAULT6
FM_7MN_DEF_DT  =   DEFAULT7
FM_8MN_DEF_DT  =   DEFAULT8
FM_9MN_DEF_DT  =   DEFAULT9
```

The entries in the next excerpt define the formats. There is a one–to–one correspondence between the substitution variables defined above and the formats defined below. (The tokens in the formats are explained in the next section.)

```
SM_0DEF_DTIME  =   %m/%d/%2y %h:%0M
SM_1DEF_DTIME  =   %m/%d/%2y
SM_2DEF_DTIME  =   %h:%0M
SM_3DEF_DTIME  =   %m/%d/%2y %h:%0M
SM_4DEF_DTIME  =   %m/%d/%2y %h:%0M
SM_5DEF_DTIME  =   %m/%d/%2y %h:%0M
SM_6DEF_DTIME  =   %m/%d/%2y %h:%0M
SM_7DEF_DTIME  =   %m/%d/%2y %h:%0M
SM_8DEF_DTIME  =   %m/%d/%2y %h:%0M
SM_9DEF_DTIME  =   %m/%d/%2y %h:%0M
```

Therefore, FM_0MN_DEF_DT defines the name of the first substitution variable, which is DEFAULT and SM_0DEF_DTIME defines its format, %m/%d/%2y %h:%0M.

Developers use the FM_ entries when creating date/time fields on screens. The FM_ entries are understood only by jxform. All runtime date/time functions use the SM_ entries. If a developer uses sm_sdtime or sm_udtime to format a date or time at runtime, the function call uses tokens, not the Screen Editor mnemonics like MONL or DEFAULT DATE.

3.6.3
The Date Time Tokens

When you specify a date format as a field edit in jxform you will use one of the date/time substitution variables. However, when specifying a format in the message file or as an argument to sm_sdtime or sm_udtime, you must use some combination of tokens. This way, **JAM** does not need to parse the message file, and the library functions may be used without knowing the names of substitution variables defined or modified in the message file. When **JAM** performs date calculations using a format, it will replace tokens with their appropriate values. All other characters in the format (i.e., commas, slashes, colons, etc.) will be used literally. The tokens are listed below. Most of these substitute numeric values; for those that substitute text, we indicate the message entries which they use.

Description	*Token*	*Message Entries for Text*
Year:		
4 digit	%4y	
2 digit	%2y (Use Setup File to specify century break)	
Month:		
numeric (1 or 2 digit)	%m	
numeric (2 digit)	%0m	
abbreviated name (3 char)	%3m	SM_MONA1 . . . SM_MONA12
full name	%*m	SM_MONL1 . . . SM_MONL12

Description	*Token*	*Message Entries for Text*
Day of the Month:		
numeric (1 or 2 digit)	%d	
numeric (2 digit)	%0d	
Day of the Week:		
abbreviated name (3 char)	%3d	SM_DAYA1...SM_DAYA7
full name	%*d	SM_DAYL1...SM_DAYL7
Day of the Year:		
numeric (1–365)	%+d	
Time:		
hour (1 or 2 digit)	%h	
hour (2 digit)	%0h	
minute (1 or 2 digit)	%M	
minute (2 digit)	%0M	
second (1 or 2 digit)	%s	
second (2 digit)	%0s	
AM and PM	%p	SM_AM, SM_PM
Default Formats:		
formats specified in message file entries[3]	%0f - %9f	SM_0DEF_DTIME to SM_9DEF_DTIME
Other:		
literal percent sign	%%	

3.6.4

Making The Changes[3]

Customizing the Default Formats

If you wish dates formatted with DEFAULT to appear like November 30, 1990 1:05 PM then change the entry

```
SM_0DEF_DTIME  =   %m/%d/%2y %h:%M0
```

to

```
SM_0DEF_DTIME  =   %*m %d, %4y %h:%M0 %p
```

and compile the file with msg2bin.

The tokens for SM_3DEF_DTIME through SM_9DEF_DTIME are the same as the token default for SM_0DEF_DTIME. These additional entries are provided so that you may create your own date/time formats. You also may rename the appropriate substitution variable. For example, change the entries FM_3MN_DEF_DT and SM_3DEF_DTIME to the following:

```
FM_3MN_DEF_DT  =   DAYOFYEAR
...
SM_3DEF_DTIME  =   %+d/%4y
```

If the date is 11/30/90 and a date/time field is formatted in jxform with DAYOFYEAR, the date will appear as

```
334/1990
```

Creating Defaults for Non–English Applications

If you are developing an application for French endusers, you should create a French version of the message file. Make a copy of the file and translate the text assigned to SM_DAYA1...SM_DAYA7, SM_DAYL1..SM_DAYL7, SM_MONA1...SM_MONA12, and SM_MONL1...SM_MONL12. For example,

```
SM_DAYA1    =   dim
SM_DAYA2    =   lun
SM_DAYA3    =   mar
...
SM_DAYL4    =   mercredi
SM_DAYL5    =   jeudi
SM_DAYL6    =   vendredi
```

3. These tokens are provided so that default formats may be used with the library functions sm_sdtime and sm_udtime.

```
SM_DAYL7    =   samedi
...
...
SM_MONA1    =   jan
SM_MONA2    =   fév
SM_MONA3    =   mar
...
SM_MONL7    =   juillet
SM_MONL8    =   août
SM_MONL9    =   septembre
SM_MONL10   =   octobre
SM_MONL11   =   novembre
SM_MONL12   =   décembre
```

If you will use DEFAULT to format date/time fields, you should modify its format according to European customs. For example,

```
SM_0DEF_DTIME  =   %d %*m %4y %h:%0M
```

and then the date specified with the format DEFAULT would appear as `30 novembre 1990 13:05`.

This method is particularly useful if you are distributing the same application to endusers who speak different languages. A user's `smvars` file or system environment can specify the names of the applicable message file and screen libraries. Date/time fields created with `jxform` will display the date in a language and format familiar to the individual end–user, but all programming code may be independent of the enduser's language.

Creating Defaults in a Non–English Version of JAM

If you localizing **JAM** for non–English speaking developers, you will translate month and day text as above, and customize formats. In addition, you will also translate the names of substitution variables. These entries are adjacent in the message file, beginning with FM_YR4 and ending with FM_9MN_DEF_DT. For example if you are translating for French–speaking developers, these entries might begin like the following,

```
FM_YR4  =   ANNÉE4
FM_YR2  =   ANNÉE2
FM_MON  =   MOIS
FM_MON  =   MOIS2
FM_DATE =   JOUR
...
```

Developers would use these substitution variables to create date/time formats in `jxform`. For example, JOUR–MOIS–ANNÉE2. Developers specifying formats for the library functions `sm_sdtime` or `sm_udtime` must use the tokens described in Section 3.6.3

3.6.5

Literal Dates in Calculations

In the message file there is also an entry for specifying the format of literal dates used in `@date` calculations. The message file entry `SM_CALC_DATE` specifies this format. By default, it is `%m/%d/%4y`. For example, `sm_calc` could be used with a literal date to count the number of days until the millennium.

```
sm_calc (0,0,'days = @date(1/1/2000)- @date(today)');
```

3.7

CURRENCY FORMATS

In Release 5, developers may create an unlimited number of currency formats on a screen. You may modify the message file to store ten default currency formats. Like the date/time message entries, an `SM_` entry defines a format, and a corresponding `FM_` entry defines the name of a substitution variable which may used in `jxform` to specify the format. See the table below.

Message Entry	*Substitution Variable*	*Corresponding Message Entry*	*Default Format*
FM_0MN_CURRDEF	= CURRENCY	SM_0DEF_CURR	= ".22,1$"
FM_1MN_CURRDEF	= NUMERIC	SM_1DEF_CURR	= ".09,"
FM_2MN_CURRDEF	= PLAIN	SM_2DEF_CURR	= ".09"
FM_3MN_CURRDEF	= DEFAULT3	SM_3DEF_CURR	= ".09"
FM_4MN_CURRDEF	= DEFAULT4	SM_4DEF_CURR	= ".09"
FM_5MN_CURRDEF	= DEFAULT5	SM_5DEF_CURR	= ".09"
FM_6MN_CURRDEF	= DEFAULT6	SM_6DEF_CURR	= ".09"
FM_7MN_CURRDEF	= DEFAULT7	SM_7DEF_CURR	= ".09"
FM_8MN_CURRDEF	= DEFAULT8	SM_8DEF_CURR	= ".09"
FM_9MN_CURRDEF	= DEFAULT9	SM_9DEF_CURR	= ".09"

3.7.1

The Formats

Currency formats have the form ***dmxtpccccc*** where

- ***d*** = decimal symbol (usually a period or comma)
- ***m*** = minimum number of decimal places
- ***x*** = maximum number of decimal places
- ***t*** = thousands' separator (i.e., a comma or period; use `b` for a blank)
- ***p*** = placement of currency symbol (`l`, `r`, or `m`)
- ***ccccc*** = currency symbol (up to 5 characters, including blank spaces)

Therefore, in the format `".22,1$"`

d =	.	Period is the decimal symbol.
m =	2	2 is the minimum number of decimal places.
x =	2	2 is the maximum number of decimal places.
t =	,	Comma is the thousands' separator.
p =	l	Currency symbol is placed on the left.
ccccc =	$	Dollar sign is the currency symbol (1 character).

3.7.2

Making Changes

For example, you may need to add a format for the French Franc. You could make the following changes to the message file:

```
SM_9DEF_CURR = ',22.r F'
.
.
.
FM_9MN_CURRDEF = FRANC
```

and compile the message file with `msg2bin`.

A number displayed with the format `CURRENCY` would appear in this form:

```
$999,999.99
```

The same number displayed with the format `FRANC` would appear in this form:

```
999.999,99 F
```

Please note that blank spaces before and after currency character(s) become a part of the currency symbol. This makes it possible to specify leading or trailing blanks in a format.

3.8
JAM DECIMAL SYMBOLS

JAM accommodates 3 types of decimal symbols. These decimals differ in scope and function.

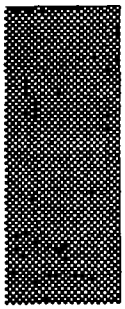
The *system* decimal symbol is the character used by the operating system when translating characters to internal values or vice versa (e.g., C routines `atof`, `sprintf`, etc.).

The *local* decimal symbol is defined by the message file entry for `SM_DECIMAL` (default = .) It can replace the system symbol within the scope of a **JAM** application. If the system and local symbols are different, **JAM** will translate appropriately when interacting with system routines. The `SM_DECIMAL` entry is useful when the system decimal symbol is inappropriate or inconvenient for application developers.

A *field* decimal symbol is defined by a currency edit for a specific field. This symbol is used only for data entry validation and the display of field values. Field decimal symbols are useful when you need to handle multiple decimal conventions within a single application. See the chapter **Writing International Applications** in the *Programmer's Guide* for more information.

3.9
USING ALTERNATE MESSAGE FILES

The `SMMSGS` environment variable specifies the file to be read into memory at initialization. If you are serving an international market, you may want to give users the option of selecting from alternate message files. At run–time the user set can the environment variable `SMMSGS` for the appropriate message file. Note that the alternative files for an application must be identical in terms of the number and sequencing of user messages. (See **Adding Messages** on page 18.)



Chapter 4

System Environment and Setup Files

4.1 INTRODUCTION

JAM supports a number of configuration or setup variables which provide a convenient way for you to control many operating parameters in the JAM run–time system and utilities. These variables may appear in the system environment, or in one of two special setup files described in this chapter.

This chapter assumes that you have some knowledge of the operating system which you are using. In particular, it assumes you know how to set environment variables. DOS users should be familiar with the command `set` and the `autoexec.bat` file. UNIX users should be familiar with the command `setenv` and shells files like `.login` or `.profile`. If you are unfamiliar with these topics you should consult your operating system documentation before proceeding. In addition, new JAM users should see the `readme` file or installation notes distributed with JAM. This file gives OS–specific examples for setting environment variables needed to run JAM. It also lists and describes the contents of all the JAM subdirectories, which is a useful reference for new users.

4.1.1 New Features for Release 5

All Release 4 setup features and variables are supported. In Release 5, however, many of these features have new names, and many variables that controlled more than one

option have been split into two or more variables. Many additional features have been added as well. In new applications, we recommend that you use the Release 5 variables. Below is a list of the Release 4 variables and their equivalents in Release 5.

Release 4	*Release 5*	*Page*
SMCHEMSGATT	EMSGATT	46
	QUIETATT	46
SMCHQMSGATT	QMSGATT	46
SMCHUMSGATT	EW_BORDSTYLE	47
	EW_BORDATT	48
	EW_DISPATT	48
SMCHFORMATTS	JW_BORDSTYLE	49
	JW_BORDATT	50
	JW_DISPATT	50
	JW_FLDATT	50
SMCHSTEXTATT	STEXTATT	46
SMDICNAME	no change	40
SMDWOPTIONS	DW_OPTIONS	52
SMEROPTIONS	ER_ACK_KEY	47
	ER_KEYUSE	47
	ER_SP_WIND	47
SMFCASE	FCASE	50
SMFEXTENSION	no change	50
SMFLIBS	no change	40

Release 4	*Release 5*	*Page*
SMINDSET	IND_OPTIONS	49
	IND_PLACEMENT	49
	SB_OPTIONS	49
SMINICTRL	no change	40
SMININAMES	no change	41
SMMPSTRING	IN_MNUSTRING	44
	IN_MNUFOLD	45
SMOKOPTIONS	IN_BLOCK	42
	IN_WRAP	44
	IN_RESET	44
	IN_VARROW	43
	IN_HARROW	42
	IN_VALID	44
	IN_ENDCHAR	43
SMUSEEXT	F_EXTSEP	51
	F_EXTREC	50
	F_EXTOPT	51
SMZMOPTIONS	ZM_SH_OPTIONS	48
	ZM_SC_OPTIONS	48

In Release 4 each setup variable had its own library function. In the current release, however, most **JAM** setup variables may be set at runtime with a single library function, `sm_option`. See the **JAM** *Upgrade Guide* for a listing.

4.2 COMMONLY USED AND REQUIRED VARIABLES

The following list summarizes the most commonly used environment variables:

- SMMSGS — File name for message text.
- SMVIDEO — File name for video information.
- SMKEY — File name for keyboard translation.
- SMVARS — File name for consolidating configuration variables.
- TERM — Terminal mnemonic.
- SMTERM — Substitute for TERM

The first three are required. They name configuration files used by **JAM** to describe its operating environment. **JAM** finds them by looking either in the system environment, or in a binary file named by SMVARS. If it fails to find either the variables or the configuration files themselves, it will post a message and exit.

The system variable TERM and the **JAM** variable SMTERM are used in with SMVARS. You may replace the three environment variables with SMVARS. This variable gives the name of a binary file containing the other screen manager variables. A typical SMVARS source file might look like the following:

```
SMKEY   =   (vt|vt950)/jam/config/vtkeys.bin
SMKEY   =   (vt100)/jam/config/vt100keys.bin
SMVIDEO =   /jam/config/vt100vid.bin
SMMSGS  =   /usr/local/msgfile.bin
SMPATH  =   /appl/masks
```

The lists enclosed in parentheses are terminal types; **JAM** uses them to find the appropriate files for your terminal. The SMVARS source file must be converted to binary using `var2bin`; the system environment then needs only the name of the binary file (and perhaps your terminal), such as

```
SMVARS= /usr/local/smvars.bin
TERM= vt100
```

The terminal type, used to match against the lists in parentheses, is taken from the variable SMTERM, or from TERM if that is not present. If you want **JAM** to recognize a terminal mnemonic different from TERM, put it in SMTERM. For example, the text editor might work fine with the terminal in VT100 emulation, but **JAM** could want the features of VT220 emulation; you could set TERM to VT100 and SMTERM to VT220.

4.2.1

JAM Initialization

Application programs initialize **JAM** by calling `initcrt`. This call must precede most library routine calls. Exceptions are calls which install memory–resident message, key and video files, or which set options. `initcrt` first calls an optional user–supplied initialization routine, which may initialize the character string `sm_term`. Main programs written in C are provided for those systems that require it, if C functions are to be used. These already call `initcrt`. Other systems permit other language main programs, but usually require a system function to be called before any C routine, including `initcrt`.

`initcrt` then looks for `SMVARS` and `SMSETUP` in the system environment, and uses them to read setup files. Subsequently, setup variables are sought first in the system environment and then in the setup files.

Next the terminal type is determined and placed in an internal character array called `sm_term`. An application program can force a terminal type by setting `sm_term` before `initcrt` is called. If the array is empty, the setup variable `SMTERM` is sought next, then `TERM`. If neither is found, initialization is attempted without a terminal type.

`SMMSGS` comes next. If this variable is not found, or `msginit` is unable to read it, **JAM** will abort initialization. Initialization errors in file I/O are reported using the C library function `perror`; these messages are system–dependent. Other errors encountered before the message file is loaded use hard–coded messages. Afterward, all error messages are taken from the message file.

Video and keyboard initialization are next attempted, in that order using `keyinit` and `vinit`. If **JAM** still cannot determine which configuration files to use, it prompts the user for a terminal type, and retries the entire sequence.

After ensuring that the environment is set up, `initcrt` initializes the operating system's terminal channel. It is set to "no echo" and non–buffered input. If other changes are desired (e.g., from 7 to 8 data bits), they can be made in the user initialization routine.

Next the initialization string found in the video file is transmitted to the terminal. The video chapter in this guide gives details; here we simply note that system calls can be embedded in the string. Often this feature can be used in place of a user initialization routine.

4.3
THE TWO SETUP FILES

You can use configuration variables by creating a text file of ***name*** = ***value*** pairs as described in the next section, and then running the `var2bin` utility to convert the file to a binary format.

There are two files in which you may place setup variables. The first is named by the system environment variable `SMVARS`. If your operating system does not support an environment, this file will be in a hard–coded location; `SMVARS` itself may not be put in a setup file. The second file is named by the `SMSETUP` configuration variable, which may be defined in the `SMVARS` file or in the system environment.

Any setup variable may occur in either file. If a variable occurs in both, the one in `SMSETUP` takes precedence. These variables may also be specified in the system environment, which takes precedence over any values found in the files, and can be used to entirely replace the files.

Typically, the `SMVARS` file will contain installation–wide parameters, while the `SMSETUP` file will contain parameters belonging to an individual or project.

4.4
INPUT FILE LINE FORMAT

Each line of the input file has the form

name = ***value***

where ***name*** is one of the keywords listed below, the equal sign is required, and ***value*** is a string or another keyword. If a line gets too long, it may be continued onto the next by placing a backslash (\) at the end. Lines beginning with a pound sign (#) are comments and are ignored by `var2bin`.

Certain variables, notably the JAM hardware configuration files—key translation and video—have values that depend on the type of terminal you are using. For those variables, there may be many entries in the input file in the form

name = (***term*** | ***term2*** | ... | ***termN***) ***value***

This signifies that the variable ***name*** uses the file called ***value*** for terminals of type ***term1***, ***term2***, etc. For example,

```
SMKEY = (ibm)              /usr/jam/config/ibmkeys.bin
SMKEY = (hp|hp2392|hpblk)  /usr/jam/config/hpkeys.bin
```

It is not necessary to give terminal names if you are only interested in one file. You may also provide, along with a number of terminal–qualified entries, one entry that is not terminal–qualified. This will serve as the default and it must be last in the list.

Variables that are terminal–dependent are noted below.

4.5
SETUP VARIABLES

Broadly speaking, setup variables fall into three classes: those that specify other configuration files, those that are essentially parameters to library routines, and those that specify defaults for file naming.

4.5.1
Configuration File Setups

SMEDITOR — Name of the text editor to use in JPL procedure windows of the screen editor when PF5 is pressed.

```
SMEDITOR= vi
```

SMKEY — Pathname of the binary file containing a key translation table for your terminal. Refer also to the `key2bin` and `modkey` utilities, the chapter on key files in this guide, and the library functions `sm_keyinit` and `sm_getkey`. This variable is terminal–dependent, and may be overridden by the system environment.

```
SMKEY= (vt100)/usr/jam/config/vt100keys.bin
```

SMLPRINT — Operating system command used to print the file generated by the local print key (LP). It must contain the string `%s` at the place where the file name should go. This variable may be overridden by the system environment. It is optional.

```
SMLPRINT= print %s
```

SMMSGS — Pathname of the binary file containing error messages and other printable strings used by the JAM run–time system and utilities. Refer also to the `msg2bin` utility, the chapter on message files in this guide, and the library functions `msg_read` and `msg_get`. This variable is terminal–dependent, and may be overridden by the system environment.

```
SMMSGS = /usr/config/msgfile.bin
```

SMPATH

List of directories in which the **JAM** runtime system should search for screens and JPL procedures. Place a vertical bar | between directory paths. No blank spaces should appear in the setup string. Refer to the library procedure `sm_r_window`. This variable is terminal–dependent, and may be overridden by the system environment. It is optional.

```
SMPATH = /usr/app/forms|/usr/me/testforms
```

SMSETUP

Pathname of one additional binary file of setup variables. This variable is terminal–dependent, and may be overridden by the system environment. It is optional.

```
SMSETUP = hpsetup.bin
```

SMVIDEO

Pathname of the binary file containing video control sequences and parameters used by the **JAM** run–time system. Refer also to the `vid2bin` utility, the video chapter in this guide, and the library function `sm_vinit`. This variable is terminal–dependent, and may be overridden by the system environment.

```
SMVIDEO = \
  (vt100|x100)/usr/jam/config/vt100vid.bin
```

SMDICNAME

Pathname of the application's data dictionary. See the library function `sm_dicname`. May be overridden in the system environment.

```
SMDICNAME = /usr/app/dictionary.dat
```

SMFLIBS

Pathname of a screen library that is to remain open while **JAM** is active. Each open library should have its own entry. See the library functions `sm_r_window` and `sm_l_open`.

```
SMFLIBS = /usr/app/genlib
SMFLIBS = /usr/me/mylib
```

SMINICTRL

May occur many times. Each occurrence binds a function key to a control string, which the **JAM** runtime system will use in the absence of a control string in the screen. To disable a JYACC–supplied default function key, bind it to a control string function that does nothing or one which calls `sm_bel`. See the library function `sm_putjctrl`.

```
SMINICTRL = PF2 = ^toggle_mode
```

SMININAMES — Supplies a list of local data block initialization file names for use by `sm_ldb_init`. It is equivalent to the library function `sm_ininames`. The file names are separated by commas, blanks, or semicolons; there may be up to ten of them. See the section on LDB initialization in the *Author's Guide* for more information on creating and formatting the files.

```
SMININAMES = tables.ini; config.ini
```

4.6
SETUPS FOR sm_input

The following variables control features of `sm_input`. There is a sample statement for each of these variables at the end of the chapter. These variables may also be changed at runtime with the library function `sm_option`.

4.6.1
Display Attributes

Many of the variables take display attributes as parameters. Here is a table of display attribute keywords:

Foreground Color	*Background Color*	*Attribute*
	B_HILIGHT	NORMAL_ATTR
BLACK	B_BLACK	BLANK
BLUE	B_BLUE	REVERSE
GREEN	B_GREEN	UNDERLN
CYAN	B_CYAN	BLINK
RED	B_RED	HILIGHT
MAGENTA	B_MAGENTA	STANDOUT
YELLOW	B_YELLOW	DIM
WHITE	B_WHITE	ACS

For a single display attribute, you may select from this table one color and any number of other attributes. In a setup file, separate the or–ed attributes with blanks, commas, or semicolons.

```
EW_BORDATT = REVERSE HILIGHT GREEN
EW_DISPATT = RED, B_WHITE; BLINK
```

With `sm_option`, use a comma to separate a variable from its attributes. Use vertical bars to separate or–ed attributes.

```
sm_option (EW_BORDATT, REVERSE | HILIGHT | GREEN);
sm_option (EW_DISPATT, RED | B_WHITE | BLINK);
```

4.6.2

Setups for User Input

The following variables may be defined in a setup file, or at runtime with `sm_option`. (D) indicates the default option for a variable.

In a setup file, the format is ***variable*** = ***option***. For example,

```
IN_BLOCK = OK_BLOCK
```

There is a statement for each variable in the sample setup file at the end of this chapter.

At runtime, the format is `sm_option` (***variable***, ***option***). For example,

```
sm_option (IN_BLOCK, OK_BLOCK);
```

You may also use `sm_option` and the option NOCHANGE to determine the runtime setting of any variable described in this section. For example,

```
sm_option (IN_BLOCK, NOCHANGE);
```

will return the current value of IN_BLOCK, either OK_NOBLOCK or OK_BLOCK. The function call will not change the value of IN_BLOCK.

Cursor Appearance and Movement

IN_BLOCK	**Set Cursor Appearance.**
OK_NOBLOCK	Cursor occupies one character position in a field. (D)
OK_BLOCK	Current field is changed to reverse video to simulate a large cursor. The cursor occupies the entire field.
IN_HARROW	**Set Horizontal Arrow Movement.**
OK_FREE	Free cursor movement.

OK_RESTRICT	The cursor moves left and right in the current field, but it does not leave the field.
OK_COLM	The cursor is positioned to the closest field on the current line.
OK_SWATH	Same as OK_COLM.
OK_NXTLINE	The cursor is positioned to the nearest field in the column closest to the current column. Wrapping is observed, if set.
OK_NXTFLD	The cursor is positioned to the field closest to the current line and column. The calculation uses the diagonal distance, assuming a 5 to 2 aspect ratio.
OK_TAB	Left–arrow backtabs to the end of the previous field, and right–arrow tabs to the first character in the next field. Wrapping is observed if set. The next and previous field edits are not observed. **(D)**
OK_TABNXT	Like OK_TAB, but the next field and previous field edits are observed.
IN_VARROW	**Set Vertical Arrow Movement.**
OK_FREE	Free cursor movement.
OK_RESTRICT	Vertical arrow keys ignored in current field.
OK_COLM	The cursor is positioned to the nearest field that overlaps the current column. Wrapping is observed, if set.
OK_SWATH	The cursor is positioned to the closet field that overlaps the swath containing the current field. Wrapping is observed if set.
OK_NXTLINE	The cursor is positioned to the nearest field whose line is closest to the current line. Wrapping is observed, if set. **(D)**
OK_NXTFLD	The cursor is positioned to the field nearest the current line and column.
OK_TAB	Down arrow tabs to the first character in next field; up arrow backtabs to last character in the previous field.The next and previous field edits are not observed. **(D)**
OK_TABNXT	Like OK_TAB, but the next field and previous field edits are observed.
IN_ENDCHAR	**Specify Treatment Of Last Character In No Auto Tab Field.**
OK_ENDWRITE	Last character in no auto tab field is repeatedly overwritten. **(D)**

OK_ENDBEEP	Terminal beeps when user attempts to overwrite last character in no auto tab field.
IN_RESET	**Set Options for Field–reset.**
	Note that IN_RESET is ignored on word–wrapped fields.
OK_NORESET	Arrow keys can enter the middle of a field. (D)
OK_RESET	When field is entered, cursor always goes to first character position, based on justification and punctuation edits.
IN_VALID	**Set Conditions For Validation On Field Exit.**
OK_VALID	Validation is performed whenever field is exited (NL, TAB, BACKTAB, arrows, etc.).
OK_NOVALID (D)	Validation is performed only when TAB or NL is pressed. Using the arrow keys to leave a field will not validate the field. (D)
IN_WRAP	**Set Options For Arrow Wrapping.**
OK_WRAP	Arrow keys wrap. Vertical arrows wrap from top to bottom. Right arrows wrap to the beginning of next line (or first line). Left arrows wrap to end of previous line (or last line). (D)
OK_NOWRAP	Arrow keys do not wrap. Terminal beeps if user tries to move the cursor past the edge of the active screen.

Menus

IN_MNUSTRING	**Set Menu Handling Options.**
OK_NOSTRING	Compare on single key. Each data key struck by the end–user is compared against the initial character of each menu selection. As soon as a match is found, the entry is selected. Therefore, if a menu contains two or more selections beginning with the same character, the second and subsequent entries cannot be selected by a data key.
OK_STRING	Compare on key sequence. Data keys are collected until the saved sequence is long enough to match one entry unambiguously. As keystrokes are collected, the cursor moves to the entry which is closest to the top and that matches the keystrokes so far. (D)

IN_MNUFOLD **Set Case–sensitivity For Menu Selections.**

OK_NOFOLD	Must match exactly. Begins matching on first character. To make a selection, the user must use case exactly as it is shown.
OK_UPPER	Allow upper–case match. Matches on the capital letter contained in the entry, rather than the first letter in the entry. The keypress may be in either case.
OK_LOWER	Allow lower–case match. Matches on the lower–case letter contained in the entry, rather than the first letter in the entry. The keypress may be in either case.
OK_UPPER_OR_LOWER	Allow either case. Begins matching on first character. This option ignores case, unless two entries begin with the same letter in different cases. (D)

IN_SEARCH **Set Search Options For Menu Selections.**

OK_ONSCREEN	Match on–screen only. Ignore entries which are off-screen because of a virtual screen or a scrolling array.
OK_ON_AND_OFFSCREEN	Match off–screen also. (D)

IN_SUBMENU **Set Submenu Options.**

OK_CLOSE	Close submenu window once selection is made. (D)
OK_LEAVEOPEN	Leave submenu window open on selection.

4.6.3

Setups for Messages

The following variables control message display. (D) indicates the default option. The form in the `smvars` file is ***variable = option***. There is a sample statement for each of these variables at the end of the chapter. Note that the BLANK attribute keyword is ignored for messages.

SMSGPOS **Set Position of Message Line.**

number	Set the position for the message line by specifying a single number (1 is the top line of the display). This variable is ignored if the terminal has a hardware status line.

SMSGBKATT **Set Background Attributes for Message Line.**

display attributes	Set message line background attribute. Default is BLACK. See Section 4.6.1 for a list of the keywords.

STEXTATT

display attributes

Set Attributes for Status Messages.

Change the default display attribute for field status text. See Section 4.6.1 for a list of the keywords. The default is STEXTATT = WHITE.[4].

QMSGATT

display attributes

Set Attributes for Query Messages.

Supply a default display attribute for sm_query messages. The default is QMSGATT = CYAN REVERSE HILIGHT. If you change this variable without setting a color, the default foreground color becomes BLACK. Please see the note on STEXTATT for more information. See Section 4.6.1 for a list of the keywords.

EMSGATT

display attributes

Set Attributes for Error Messages.

Change the display attributes of messages displayed with sm_emsg and sm_err_reset, and the tag portion of sm_quiet_err and sm_qui_msg messages. By default, the tag portion is "ERROR:". This is from the message file entry SM_ERROR. The default is EMSGATT = WHITE BLINK HILIGHT.

If you change this variable without setting a color, the default foreground color for error messages becomes BLACK. Please see the note on STEXTATT for more information. See Section 4.6.1 for a list of the keywords.

QUIETATT

display attributes

Set Attributes for "Quiet" Error Messages.

Change the display attributes of messages displayed with sm_quiet_err and sm_qui_msg messages. Default is WHITE. See EMSGATT for changing the attributes of the "ERROR:" tag which is used with these messages.

If you change this variable without setting a color, the default foreground color for these messages becomes BLACK. Please see the note on STEXTATT for more information. See Section 4.6.1 for a list of the keywords.

The following three variables control error message acknowledgement.

4. If you change the attributes but do not specify a color, the default color becomes BLACK. For instance, if you used the entry STEXTATT = BLINK, **JAM** will display status messages with the foreground attributes BLINK and BLACK. If you were using the default message line background (see SMSGBKATT), status messages would not be visible because they would be black text on a black background. To avoid this, we recommend that you always specify a foreground or background color when setting attributes text. If this is not convenient, you may set the variable SMSGBKATT to a color other than BLACK.

ER_ACK_KEY

Define Error Acknowledgement Key.

key

There are no keywords—the value must be specified explicitly. The key can be given as number (in decimal, hex, or octal) representing an ASCII char, as an ASCII mnemonic (SP, SOH, ETX, etc.), as quoted character ('.', '_', etc.), or as a logical key defined in smkeys.h. The default is ' ', the space key. If you define a value other than the space bar, please see ER_SP_WIND below.

ER_KEYUSE

Use or Discard Key in sm_err_reset

ER_NO_USE

All error messages must be acknowledged by ER_ACK_KEY, which is discarded. Any other keys struck between the time of the message display and the pressing of the acknowledgment key are also discarded. By default, if the user does not press ER_ACK_KEY, **JAM** displays an error window. See ER_SP_WIND. **(D)**

ER_USE

Any keypress acknowledges an error message. The type–ahead buffer is flushed when the message is displayed, and the acknowledging keypress is saved for data–entry. Since any keypress clears the error message, the "Please press the space bar" is not used. If you set this as the default, you can still force the user to acknowledge selective messages by putting %Md at the beginning of the message text. See the Message File chapter for more information.

ER_SP_WIND

Remind User to Acknowledge Message.

ER_YES_SPWIND

If ER_KEYUSE= ER_NO_USE, and the user presses another key when ER_ACK_KEY is expected, a window appears. The default message is "Please hit the space bar after reading this message" from the message file entries SM_P1 and SM_P2. If you are using this option and a key other than the space bar for message acknowledgement, modify the message file entry SM_SP1. **(D)**

ER_NO_SPWIND

If ER_KEYUSE= ER_NO_USE, and the user presses another key when ER_ACK_KEY is expected, the terminal beeps (by calling sm_bel). A "visual" bell may be used, if the video file file has a BELL entry.

EW_BORDSTYLE

Set Border Style of Error Windows.

NOBORDER

No border.

0-9	A number between 0 and 9 indicates a style. Default is 0.
EW_BORDATT	**Set Border Attributes of Error Windows.**
display attributes	Default is HILIGHT REVERSE BLUE. See Section 4.6.1 for a list of the keywords.
EW_DISPATT	**Set Text Attributes in Error Windows.**
display attributes	Default is WHITE. See Section 4.6.1 for a list of the keywords.

4.6.4

Shifting, Scrolling, and Zooming Setups

If you are altering these defaults at runtime, call sm_option before calling the window.

ZM_SC_OPTIONS	**Set Zoom Scroll Options.**
ZM_NOSCROLL	No scroll expansion on arrays.
ZM_SCROLL	Scroll the current array and display as many occurrences as possible.
ZM_PARALLEL	Scroll all parallel or synchronized arrays. Display as many occurrences as possible. **(D)**
ZM_1STEP	Scroll and shift in one step.
ZM_SH_OPTIONS	**Set Zoom Shift Options.**
ZM_NOSHIFT	No shift expansion. Fields will shift, but no horizontal zooming will take place.
ZM_SCREEN	Shifting arrays will have as many on–screen elements as the previous form, which is the original form if ZM_SC_OPTIONS = ZM_NOSCROLL is used. Otherwise, ZM_SCREEN will display as many items as possible. All synchronized arrays are shifted together. **(D)**
ZM_ELEMENT	Show one element in the shift window, but permit scrolling with the arrow keys. If ZM_SCROLL is also selected, the shift window may be zoomed again to show all elements. If the array is synchronized, only the current array zooms.
ZM_ITEM	Show one element in the shift window. There is no way to see more occurrences.

IND_OPTIONS	**Set Shift/Scroll Indicator Options.**
IND_NONE	No indicators.
IND_SHIFT	Shift indicators only.
IND_SCROLL	Scroll indicators only.
IND_BOTH (D)	Shift and scroll indicators.
SB_OPTIONS	**Set Scroll Options for Virtual Windows.**
SB_NONE	No scroll bars or corner arrows.
SB_BARS	Show scroll bars. (D)
SB_CORNERS	Show corner arrows.
IND_PLACEMENT	**Set Position of Shift and Scroll Indicators.**
IND_FULL	Full width of field. (D)
IND_FLDENTRY	Left or right corner, according to the field's justification.
IND_FLDLEFT	Left corner of field.
IND_FLDCENTER	Center of field.
IND_FLDRIGHT	Right corner of field.
ZW_BORDSTYLE	**Set Border Style for Zoom Windows.**
NOBORDER	No border.
0-9	A number between 0 and 9 indicates a style. Default is 1.
ZW_BORDATT	**Set Border Attributes For Zoom Windows.**
display attributes	Default is RED HILIGHT. See Section 4.6.1 for a list of the keywords.

4.6.5

Setups for JAM Windows

These variables control the display of the JAM "Go to:" and "Enter system command:" windows, if the screens are memory–resident. By default, JAM system screens are accessed from libraries on disk. To make these screens memory–resident, you must modify `jmain.c` and `jxmain.c`. See the *Programmer's Guide* for more information.

JW_BORDSTYLE	**Set Border Display of JAM Windows.**
NOBORDER	No border.
0-9	A number between 0 and 9 indicates a style. Default is 0.

JW_BORDATT — **Set Border Attributes of JAM Windows.**

display attributes — Default is HILIGHT REVERSE BLUE. See Section 4.6.1 for a list of the keywords.

JW_DISPATT — **Set Text Attributes of JAM Windows.**

display attributes — Default is CYAN. See Section 4.6.1 for a list of the keywords.

JW_FLDATT — **Set Field Attributes for JAM Windows.**

display attributes — Default is YELLOW HILIGHT UNDERLN. See Section 4.6.1 for a list of the keywords.

4.6.6

Setups for File Names and Extensions

The following variables control default file extensions. In Release 4, the defaults for file extensions were set with the variable SMUSEEXT. This variable is supported for backwards compatibility.

(D) indicates the default option. The form in the smvars file is ***variable = option.*** There is a sample statement for each of these variables at the end of the chapter.

FCASE — **Set Case Sensitivity for File Name Searches.**

CASE_INSENS — **JAM** ignores case when searching for a file named in a control string.

CASE_SENS — Filename searches are case sensitive. **(D)**

SMFEXTENSION — **Specify Screen File Extension.**

extension — Screen file extension is used by the **JAM** run–time system and various utilities. The default is system–dependent; it may be jam or none. If an extension is given, **JAM** will append (or prefix) it to any screen name that does not already contain an extension. Use F_EXTSEP to specify a character which will separate a filename and the extension. Use F_EXTOPT to specify the placement.

F_EXTREC — **Recognize Screen and Utility I/O File Extensions.**

FE_IGNORE — Ignore extensions. **(D)**

FE_RECOGNIZE — Recognize extensions.

F_EXTOPT

Placement of Screen and Utility I/O File Extensions.

FE_FRONT — Put the extension before the filename.

FE_BACK — Put the extension after the filename. (D)

F_EXTSEP

Specify Screen and Utility I/O File Extension Separator.

character — There are no keywords—the value must be specified explicitly. A separator character may be given as a number (in decimal, hex, or octal) which represents an ASCII char, as an ASCII mnemonic (SOH, ETX, etc.), or as quoted character ('.', '_', etc.).

4.6.7

Setups for Group Attributes

These variables control the attributes of the cursor and selected items in groups that do not use the checkbox edit.

GA_CURATT

Set Group Cursor Attributes

display attributes — Assign the desired attributes for the occurrence under the cursor. These attributes are added to those already assigned to the occurrence. This defaults to BLINK.

GA_CURMASK

Mask Group Cursor Attributes

display attributes — Mask any attributes that should not be added to the cursor attributes. If you are assigning a color as the cursor attribute, add NORMAL_ATTR to GA_CURMASK. Attributes of the occurrence will be used if they are not masked out.

GA_SELATT

Set Selected Group Occurrence Attributes

display attributes — Assign the desired attributes for a selected group occurrence. These attributes are added to those already assigned to the occurrence. This defaults to REVERSE.

GA_SELMASK

Mask Selected Group Occurrence Attributes

display attributes — Mask any attributes that should not be added to the attributes for the selected group occurrence. If you are assigning a color to GA_SELATT, add NORMAL_ATTR to

GA_SELMASK. Attributes of the occurrence will be used if they are not masked out.

4.6.8

Miscellaneous Setups

These may also be set at runtime with sm_option.

DA_CENTBREAK

two digit number

Set Default Century for 2 Digit Dates

Use this option to specify the breaking year between the twentieth and twenty–first centuries when **JAM** formats two–digit years to four digit years. By default, **JAM** assumes that all two digit years are in the twentieth century, This option allows you to specify that all two digit years less than the number specified should be in the twenty–first century. For example, if you specify 45, then all two digit years between 00 and 44 will indicate 2000 – 2044, and all two digit years between 45 and 99 will indicate 1945 – 1999.

DW_OPTIONS

Set Delayed–write Options.

DW_ON

Turns on delayed–write. Output from library functions is not sent immediately to the display, but is used to update the image in memory. When it is necessary to update the display (e.g., when the keyboard is opened), output is sent to the display one line at a time, and a check is made for keyboard input between each line. If the user presses a key before the update is completed, the key is processed before the remaining lines are displayed. This option makes **JAM** more responsive, especially at low baud rates. You may force the display of a delayed–write with the library function sm_flush. **(D)**

DW_OFF

Turns off delayed–write. The display will not be flushed until the keyboard is opened. **JAM** will not check for input while writing to the display. This option may be useful when debugging an application. If you use this option, you may need to add the entry BUFSIZ to the video file.

ENTEXT_OPTION

Set Screen Entry/Exit Processing Option.

LDB_FIRST

LDB is examined first for the value of a variable. Default on screen entry functions.

FORM_FIRST	Screen is examined first for the value of a variable. Default on screen exit functions.
EXPHIDE_OPTION	**Set Screen Expose/Hide Option.**
OFF_EXPHIDE	Process screen entry or exit functions only when screen is explicitly opened or closed. This option is the default for backwards compatibility. (D)
ON_EXPHIDE	Process screen functions when screen is explicitly opened or closed, when screen is exposed by closing an overlying window, or when screen is hidden by opening an overlying window. This option is recommended for new applications.
SK_NUMATT	**Set the Display Attributes of Soft Key Numbers**
display attributes	These are the attributes for the numbers that appear to the left of each key in a keyset. For a list of valid display attributes, see section 4.6.1 To turn off number display, set SK_NUMATT = BLANK.

4.7 BLOCK MODE OPTIONS

The following control options for block mode. They may be set at runtime with sm_option. Use sm_option (***variable***, NOCHANGE) at runtime to determine the current value of a block mode option.

BLK_MENUS	**Set Menu Behavior in Block Mode.**
BLK_BLKMENU	Menu field is unprotected, so that a field may be entered by tabbing. User selects a field by moving the cursor and pressing XMIT. For submenus, if cursor is moved outside the submenu window and XMIT is pressed, the main menu choice closest the cursor is selected.
BLK_CHARMENU	**JAM** switches to character mode when processing a menu, and switches back when menu is finished. Do not use this option if switching will cause the screen to clear.
BLK_FLDMENU	**JAM** creates fields to the left of each menu selection, space permitting. A selection is made by typing the first non–blank character of a menu selection in one of these

fields and pressing XMIT. If no data key is entered in a field, the cursor position selects the current field when XMIT is pressed, in either the submenu or the main menu. Selection by letter in a submenu applies only to the submenu.

BLK_GROUPS — **Set Group Behavior in Block Mode.**

BLK_BLKGROUP — Group fields are unprotected, so that a field may be entered by tabbing. User selects a field by moving the cursor and pressing XMIT.

BLK_FLDGROUP — **JAM** creates fields to the left of each group selection, space permitting. Typing any blank character in the field next to an occurrence selects the occurrence. If no character is typed in the field, pressing XMIT selects the occurrence where the cursor is positioned.

BLK_ERRORS — **Set Options for Error and Query Messages in Block Mode.**

BLK_BLKERR — sm_err_reset works in this mode. Messages are acknowledged with XMIT, rather than the ER_ACK_KEY. With sm_query_msg, XMIT is "yes" and EXIT is "no".

BLK_CHARERR — **JAM** switches the terminal into interactive mode before displaying an error or query message, and switches back to block mode on completion.

BLK_FLDQUERY — **JAM** remains in block mode for error messages, and creates a one–character field for query_msg. The field defaults to the value of message entry SM_YES ("y" by default). The user may change this to "n" (i.e., value of SM_NO). XMIT is used to select an answer; EXIT is equivalent to "no", regardless of the field's contents.

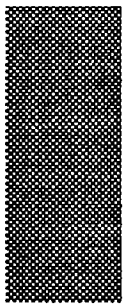
4.8
SAMPLE SETUP FILE

The following sample file illustrates the syntax for setting all of the variables discussed above.

```
SMKEY = (vt100 | x100) /usr/jam/config/vt100keys.bin
SMLPRINT = print %s
SMMSGS = /usr/config/msgfile.bin
SMPATH = /usr/app/forms|/usr/me/testforms
SMSETUP = hpsetup.bin
```

```
SMVIDEO =  (vt100 | x100)/usr/jam/config/vt100vid.bin
SMDICNAME = /usr/app/dictionary.dat
SMFLIBS = /usr/app/genlib
SMFLIBS = /usr/me/mylib
SMINICTRL= PF2 = ^toggle_mode
SMINICTRL = PF3 = &popwin(3,28)
SMINICTRL = XMIT = ^commit all
SMININAMES = tables.ini; config.ini

IN_BLOCK = OK_NOBLOCK
IN_WRAP = OK_WRAP
IN_RESET = OK_NORESET
IN_ENDCHAR = OK_ENDWRITE
IN_VALID = OK_VALID
IN_VARROW = OK_FREE
IN_HARROW = OK_TAB
IN_MNUSTRING = OK_STRING
IN_MNUFOLD = OK_LOWER
IN_SEARCH = OK_ONSCREEN
IN_SUBMENU = OK_LEAVEOPEN

ER_ACK_KEY = PF12
ER_KEYUSE = ER_NO_USE
ER_SP_WIND = ER_YES_SPWIND
EW_BORDSTYLE = 4
EW_BORDATT = REVERSE, GREEN
EW_DISPATT = YELLOW
EW_FLDATT = CYAN

EMSGATT = RED; RED, REVERSE
QUIETATT = CYAN, HILIGHT, BLINK
STEXTATT = WHITE REVERSE
QMSGATT = CYAN REVERSE
SMSGBKATT = RED
SMSGPOS = 25

ZM_SH_OPTIONS = ZM_ITEM
ZM_SC_OPTIONS = ZM_SCROLL
IND_OPTIONS = IND_BOTH
IND_PLACEMENT = IND_FLDRIGHT
SB_OPTIONS = SB_CORNERS
ZW_BORDSTYLE = 8
ZW_BORDATT = MAGENTA

JW_BORDSTYLE = 4
JW_BORDATT = HILIGHT REVERSE BLUE
JW_DISPATT = YELLOW
JW_FLDATT = WHITE HILIGHT UNDERLN


SMFEXTENSION = jam
F_EXTREC = FE_RECOGNIZE
F_EXTOPT = FE_BACK
F_EXTSEP = '.'
```

```
DW_OPTIONS = DW_OFF
ENTEXT_OPTION = LDB_FIRST
EXPHIDE_OPTION = ON_EXPHIDE
FCASE = CASE_SENS
```



Chapter 5
Video File

5.1
INTRODUCTION

JAM is designed to run on many displays with widely differing characteristics. These characteristics greatly affect **JAM**'s display of screens and messages. For example, some displays are 80 columns wide, while others have 132 columns. Similarly, the control sequences used to position the cursor and highlight data on the display are often different from model to model. **JAM** obtains display characteristics from a video file.

5.1.1
How to Use this Chapter

This chapter has two purposes. The first is to explain the entries in the **JAM** video file and the concepts used in interpreting them. Although you may never need to modify or construct a video file, you may wish to know what it does. The second purpose is to provide instructions for modifying existing video files or constructing new ones to handle new terminal characteristics.

The easiest way to create a video file is to use one of the many supplied with **JAM**. You can modify it, if you can determine that your terminal is similar. This is very often possible because so many terminals emulate others. If your system has a *terminfo* or *termcap* database, you can use the `term2vid` utility to make a functional video file from that information. Finally, if you must start from scratch, you should start with the minimal subset defined in this guide, and add entries one at a time.

5.1.2

Why Video Files Exist

Differences among terminal characteristics do not affect programs that are line oriented. They merely use the screen as a typewriter. Full–screen editors, like *emacs* or *vi*, use the screen non–sequentially; they need terminal–specific ways to move the cursor, clear the screen, insert lines, etc. For this purpose the *termcap* database, and its close relative *terminfo*, were developed. Although closely associated with UNIX, *termcap* and *terminfo* are also used on other operating systems. They list the idiosyncrasies of many types of terminals.

Text editors use visual attributes sparingly, if at all. Thus *termcap* contains minimal information about handling them. Usually there are entries to start and end "stand–out" and sometimes entries to start and end "underline." Notably missing are entries explaining how to combine attributes (i.e., reverse video and blinking simultaneously). *terminfo* can combine attributes; in practice, unfortunately, the appropriate entries are usually missing.

JAM makes extensive use of attributes in all combinations, and supports color. Rather than extending *termcap* with additional codes, which might conflict with other extensions, JYACC decided to use an independent file to describe the terminal specific information. Furthermore, some machines, notably the PC, do not have *terminfo* capability.

termcap uses a limited set of commands; notably missing are conditionals. *terminfo* uses an extensive set of commands, but the resulting sequences are excessively verbose (103 characters for the ANSI attribute setting sequence without color). Therefore, JYACC developed a set of commands that extend both *termcap* and *terminfo*. Both syntaxes are supported with only minor exceptions. All the commands needed in the video file can be written using *terminfo* syntax; many can be written using the simpler *termcap* syntax and a few can benefit by using the extended commands.

A summary of the commands used to process parameters is described in this guide; details and examples follow. Refer to those sections if you have trouble understanding the examples elsewhere in the manual.

5.1.3

Text File Format

The video file is a text file that can be created using any text editor. It consists of many instructions, one per line. Each line begins with a keyword, and then has an equal sign (=). On the right of the equal sign is variable data depending on the keyword. The data

may be a number, a list of characters, a sequence of characters, or a list of further instructions.

Comments can be entered into the file by typing a hash # as the first character of the line; that line will be ignored by `vid2bin`. All the video files distributed by JYACC are documented with comments.

It is essential that the instruction formats listed in this guide be followed closely. For efficiency, no error checking is done at runtime. The `vid2bin` utility checks for errors like missing, misspelled, and superfluous keywords, but not for duplicated or conflicting entries.

5.1.4
Minimal Set of Capabilities

The only required entries in the video file are for positioning the cursor (`CUP`) and erasing the display (`ED`).

In the absence of other entries, **JAM** will assume a 24–line by 80–column screen. The 24th line will be used for status text and error messages, and the remaining 23 will be available for screens. It will assume that no attributes are supported by the terminal. Since non–display is supported by the software, that attribute will be available. The underline attribute is simulated by underscores placed wherever blanks appear in an underlined field. Clearing a line will be done by writing spaces. Borders will be available, and will consist of printable characters only.

Although **JAM** will function with those two entries, it will have limited features. The most glaring shortcoming will be the lack of visual attributes. Speed may also be a problem, since the sole purpose of many entries in the video file is to decrease the number of characters transmitted to the terminal.

5.1.5
A Sample Video File

The following video file is for a basic ANSI terminal, like a DEC VT100.

```
# Display size (these are actually the default # values)
LINES   = 24
COLMS   = 80

# Erase whole screen and single line
ED  = ESC [ 2 J
EL  = ESC [ 0 K
```

```
# Position cursor
CUP = ESC [ %i %d ; %d H

# Standard ANSI attributes, four available
LATCHATT    = REVERSE = 7 UNDERLN = 4 BLINK = 5 HILIGHT = 1
SGR = ESC [ 0 %u %5(%t ; %c %; %) m
```

This file contains the basic capabilities, plus control sequences to erase a line and to apply the reverse video, underlined, blinking, and highlighted visual attributes. The entries for CUP and SGR are more complicated because they require additional parameters at run–time. The percent commands they contain are explained later.

5.1.6

An MS–DOS Video File

By default, JAM displays data on the console by directly accessing the PC's video RAM. On machines that are not 100% IBM–compatible, it will use BIOS calls instead. Use the entry INIT = BIOS for these machines. Under no circumstances does JAM use DOS calls or the ANSI.SYS driver. Video files for both monochrome and color displays are distributed with JAM.

Because JAM contains special code for the PC display, most of the entries that contain control sequences are irrelevant, and are given a value of PC in the distributed video files. You should leave these entries alone, since their presence is required but their values are irrelevant. Entries that do not contain control sequences, such as LINES, GRAPH, and BORDER, can be changed as usual. A sample PC video file follows.

```
LINES = 25
COLMS = 80

# INIT and RESET can change the cursor style if desired
# INIT = C topscan, bottomscan [, flag]
# RESET = C topscan, bottomscan [, flag]
# where topscan is top scan line (0 - 7), bottom scan is
# the bottom scan line (0 - 7) and flag is 0 for fast
# blinking, 1 for no cursor, 2 for fast blinking (again)
# and 3 for slow blinking

# other flags are available in the INIT sequence:
# BIOS means use BIOS (ROM) calls for display rather than
# writing directly to video memory
# XKEY means use services 0x10 and 0x11 for keyboard
# access (INIT 10) rather than 0 and 1. This allows use
# of F11 and F12 on those systems that support it.
# GRAYKEYS means distinguish between gray and white cursor keys.
# MULTISHIFT permits combination shifts like Ctrl-Alt and Shift-Alt.
# WINDOWS allows JAM to move invisible cursors in DOS windows
# running under Windows.
# RETRACE means wait for retraces before writing to the
```

```
# video buffer. This slows writing but reduces snow on
# CGAs.
#
# the default for init is:
# INIT = C 0,7,2
# if there is no RESET sequence, the cursor is restored
# to its style before the formaker routine was entered

# most sequences are handled by assembler functions,
# these should be set to "PC" to function correctly

# erase display and erase to end of line
ED = PCEL = PC

# cursor on and cursor off
CON = PC
COF = PC

# support insert mode cursor style
INSON = C 6,7,0
INSOFF = C 0,7,0

# absolute and relative cursor positioning
CUP = PC
CUU = PC
CUD = PC
CUB = PC
CUF = PC

# color should be specified as shown, attributes simply
# listed
COLOR = BLUE = 1 GREEN = 2 RED = 4 BACKGRND
LATCHATT = HILIGHT BLINK
SGR = PC

# erase window
EW = PC

# save and restore cursor position and attributes
SCP = PCRCP = PC

# repeat character
REPT = PC

# PC graphics characters are used for borders and shift
# arrows

#        __   __   __
#       |           |   |     |     |           |
#       |           |   |     |     |__   __    __|


BORDER =   SP  SP  SP  SP  SP  SP  SP  SP   \
    0xda    0xc4    0xbf    0xb3    0xb3    0xc0    0xc4    0xd9 \
    0xc9    0xcd    0xbb    0xba    0xba    0xc8    0xcd    0xbc \
    0xd5    0xcd    0xb8    0xb3    0xb3    0xd4    0xcd    0xbe \
    0xd6    0xc4    0xb7    0xba    0xba    0xd3    0xc4    0xbd \
```

```
    0xdc    0xdc    0xdc    0xdd    0xde    0xdf    0xdf    0xdf \
    .   .   .   :   :   :   .   .     \
    0xb0    0xb0    0xb0    0xb0    0xb0    0xb0    0xb0    0xb0 \
    0xb2    0xb2    0xb2    0xb2    0xb2    0xb2    0xb2    0xb2 \
    0xbd    0xbd    0xbd    0xbd    0xbd    0xbd    0xbd    0xbd

#   ___             |
#   |   |-  -|- -|   ---
BOX =   SP  SP  SP  SP  SP \
    0xc2    0xc3    0xc5    0xb4    0xc1 \
    0xcb    0xcc    0xce    0xb9    0xca \
    0xd1    0xc6    0xd8    0xb5    0xcf \
    0xd2    0xc7    0xd7    0xb6    0xd0 \
    0xb1    0xb1    0xb1    0xb1    0xb1 \
    0xf9    0xf9    0xf9    0xf9    0xf9 \
    0xb0    0xb0    0xb0    0xb0    0xb0 \
    0xb2    0xb2    0xb2    0xb2    0xb2 \
    0xdb    0xdb    0xdb    0xdb    0xdb
#               ^       ^
#   <-  ->  <-> |   |   |
#                   v   v
ARROWS = 0x1b 0x1a 0x1d 0x18 0x19 0x12

# update the cursor position display every .10 seconds
CURPOS = 1

# all PC graphics are available
GRTYPE = PC
```

Here the `INIT` entry specifies the cursor style. See Section 5.4.1 for more information.

5.2
VIDEO FILE FORMAT

All white space (spaces and tabs) is skipped, except where noted below. A logical line may be continued to the next physical line by ending the first line with a backslash. Do not leave a space between the backslash and carriage return. To enter a backslash as the last character of the line, use two backslashes (without spaces). Thus

`text \`	means a continuation line
`text \\`	ends with a backslash
`text \\\`	has a backslash and a continuation.

A double quote " starts a string. The quote itself is skipped. Text between it and the next double quote (or the end of the line) is taken literally, including spaces. To include a double quote in a quoted string, use backslash quote `\"` with no space between. For example,

`"stty tabs"`	has an embedded space
`stty tabs`	does not.

The percent sign is a control character. To enter a literal percent sign, you must double it (i.e., `%%`).

There are 3 ways to put non–printing characters, like control characters, in the video file:

- Any character at all can be entered as `0x` followed by two hexadecimal digits. For example, `0x41` can be used for `A`, or `0x01` for control–A, etc. This method is particularly useful for entering codes in the range `0x80` to `0xff`.
- Control characters in the range `0x01` to `0x1f` can be represented by a caret `^` followed by a letter or symbol. Either `^A` or `^a` can represent `SOH (0x01)`. The symbols are `^[` for `ESC`; `^\` for `FS`; `^]` for `GS`; `^^` for `RS`; and `^_` for `US`.
- More control characters can be represented by two– or three–character ASCII mnemonics. This method is particularly useful for entering control sequences to the terminal, since the manuals often list such sequences using mnemonics. Here is a list:

Mnemonic	*Hex*	*Mnemonic*	*Hex*	*Mnemonic*	*Hex*	*Mnemonic*	*Hex*
NUL	0x00	DLE	0x10			DCS	0x90
SOH	0x01	DC1	0x11			PU1	0x91
STX	0x02	DC2	0x12			PU2	0x92
ETX	0x03	DC3	0x13			STS	0x93
EOT	0x04	DC4	0x14	IND	0x84	CCH	0x94
ENQ	0x05	NAK	0x15	NEL	0x85	MW	0x95
ACK	0x06	SYN	0x16	SSA	0x86	SPA	0x96
BEL	0x07	ETB	0x17	ESA	0x87	EPA	0x97
BS	0x08	CAN	0x18	HTS	0x88		
HT	0x09	EM	0x19	HTJ	0x89		
NL	0x0a	SUB	0x1a	VTS	0x8a		
VT	0x0b	ESC	0x1b	PLD	0x8b	CSI	0x9b
FF	0x0c	FS	0x1c	PLU	0x8c	ST	0x9c
CR	0x0d	GS	0x1d	RI	0x8d	OCS	0x9d
SO	0x0e	RS	0x1e	SS2	0x8e	PM	0x9e
SI	0x0f	US	0x1f	SS3	0x8f	APC	0x9f
SP	0x20	DEL	0x7f				

The rightmost two columns are extended ASCII control codes, which can be transmitted only if the communication line and terminal use eight data bits. If this is not possible, the 8–bit code may be replaced by two 7–bit codes. The first code is ESC (0x1b), the second 0x40 less than the desired 8–bit control character. For example, CSI (0x9b) would be replaced by ESC 0x5b, or ESC [. If a video file contains extended ASCII control codes, **JAM** will assume they can be used; it will not transmit the two–character sequence automatically.

Note: PRIME computers, and some others, internally toggle the high bit of a character; ESC on a PRIME is 0x9b and CSI is 0x1b, not vice versa. The numbers given in this document are always standard ASCII.

5.2.1

Keyword Summary

All the video file entry keywords are listed here, arranged by function. Subsequent sections explain each one in detail.

Basic Capabilities (page 77)

LINES	Number of lines on screen
COLMS	Number of columns on screen
INIT	Initialization sequence
RESET	Undo initialization sequence
REPT	Repeat following character sequence
REPMAX	Maximum number of repeated characters
BOTTOMRT	Last position of screen may be written without scrolling the display
BUFSIZ	Number of characters to accumulate before flushing
KBD_DELAY	Timing interval for keyboard input

Erasure Commands (page 79)

ED	Erase entire display
EL	Erase to end of current line
EW	Erase window

Cursor Position (page 80)

CUP	Absolute cursor position
CUU	Cursor up
CUD	Cursor down
CUF	Cursor forward
CUB	Cursor backward
CMFLGS	Allowed cursor–motion shortcuts

Cursor Appearance (page 81)

CON	Turn cursor on

COF	Turn cursor off
SCP	Save cursor position and attribute
RCP	Restore cursor position and attribute
INSON	Insert–mode cursor
INSOFF	Overstrike–mode cursor

Display Attributes (page 82)

SGR	Set graphics rendition (latch)
ASGR	Set graphics rendition (area)
LATCHATT	List of available latch attributes
AREAATT	List of available area attributes
ARGR	Remove area attribute
SPXATT	List of attributes that do not affect space
COLOR	List of colors

Message Line (page 90)

OMSG	Open message line
CMSG	Close message line
MSGATT	Message line attributes

Soft Key Labels (page 91)

KPAR	Soft key label description
KSET	Load soft key labels
KSON	Turn keyset labels on
KSOFF	Turn keyset labels off

Graphics (page 93)

MODE0	Normal character set sequence
MODE1	Locking shift to alternate character set 1
MODE2	Locking shift to alternate character set 2
MODE3	Locking shift to alternate character set 3

MODE4	Non–locking shift to alternate character set 1
MODE5	Non–locking shift to alternate character set 2
MODE6	Non–locking shift to alternate character set 3
GRAPH	Graphics character equivalents
GRTYPE	Shortcut for defining graphics characters

Borders and Line Drawing (page 95)

BORDER	Characters that make up the 10 border styles
BRDATT	Available border attributes
BOX	Characters that make up the styles for box and line drawing

Indicators (page 98)

ARROWS	Indicator characters for shifting and scrolling
BELL	"Visible bell" alarm sequence
CBSEL	Selection character for groups with box edits
CBDSEL	Deselection character for groups with box edits

Drivers (page 99)

MOUSEDRIVER	Name of mouse driver
BLKDRVR	Name of block mode driver

Miscellaneous (page 99)

CURPOS	Display the current cursor position on the status line
COMPRESS	Output data compression for *Jterm*

5.3 PARAMETERIZED CHARACTER SEQUENCES

Certain control sequences cannot be completely specified in advance. An example is the cursor position sequence, which requires the line and column number before moving. The commands using these sequences must be passed extra parameters.

The number in parentheses is the number of parameters for each keyword.

Keyword	*Action*	*Parameters*
REPT	repeat sequence (2)	character number of times to repeat
EW	erase window (5)	start line start column number of lines number of columns background color
CUP	cursor position (2)	line and column (relative to 0)
CUU	cursor up (1)	line increment
CUD	cursor down (1)	line increment
CUF	cursor forward (1)	column increment
CUB	cursor backward (1)	column increment
SGR	set latch graphics rendition (12)	see Section 5.4.5
ASGR	set area graphics rendition (12)	see Section 5.4.5

5.3.1

Summary of Percent Commands

Parameters are encoded in sequences by percent commands, sequences starting with the `%` symbol. This is superficially similar to the way the C library function `printf` handles parameters. Some percent commands cause data to be output; others are used for control purposes. Every parameter that is to be output requires a percent command. **JAM** uses a stack mechanism as does *terminfo*; it is described in the next section. Percent commands are summarized in the list that follows. Examples and more complete descriptions are in subsequent sections.

Percent signs must be used with care, since all sequences go through the same processing, even those that do not use runtime arguments. In particular, to enter a percent sign as a literal, you must use `%%`.

In the lists below, commands are tagged to indicate their source. The tags are the following:

(C)	*termcap*
(I)	*terminfo*
(E)	JYACC extended command

Output Commands

%%	Output a percent sign (C and I)
%.	Output a character (C)
%c	Output a character (I)
%d	Output a decimal (C and I)
%#d	Output a #–digit decimal, blank filled (I)
%0#d	Output a #–digit decimal, zero filled, like the *termcap* %2 which is not supported (I)
%+	Add and output a character (C)
%#z	Output # (decimal number) binary zeros (E)
%#w	Wait (sleep) # seconds (E)
%S	Issue a system command (E)

Stack Manipulation and Arithmetic Commands

%p#	Push parameter # (1 – 11 allowed) (I)
%'c'	Push the character constant *c* (I)
%{#}	Push the integer constant # (I)
%+	Add (I)
%–	Subtract (I)
%*	Multiply (I)
%/	Divide (I)
%m	Modulus (I)
%|	Bitwise OR (I)
%^	Bitwise exclusive OR (I)
%&	Bitwise AND (I)
%=	Logical EQUAL TO (I)
%>	Logical GREATER THAN (I)
%<	Logical LESS THAN (I)

%!	Logical NOT (I)
%~	One's complement (I)

Parameter Sequencing and Changing Commands

%#u	Discard # parameters (E)
%#b	Back up # parameters (E)
%i	Increment the next two parameters (C and I)
%r	Reverse the next two parameters (C)

Control Flow Commands

%? *expr* %t *then-part* %e *else-part* %;
Conditionally execute one of two command sequences (I)

***expr* %t *then-part* %e *else-part* %;**
Same effect as previous (E)

%#(... %)	Repeat the sequence # times (E)
%l(... %)	Select operations from a list (E)

Terminfo Commands Not Supported

%P, %g	Letter variables
$<#>	Padding (use %#z instead)

5.3.2

Automatic Parameter Sequencing

In processing a keyword, a stack is used which is initially empty (and can hold up to four items); the parameters are kept in a separate list. Parameters are generally pushed on the stack as needed. The parameters are ordered and a pointer initially points to the first one. The stack is four levels deep; anything pushed off the end is lost. There are commands that push a parameter or constant onto the stack, but no explicit pop commands. Output commands transmit the value on top of the stack, then remove it. Arithmetic and logical operations take one or two operands from the top of the stack, and replace them with one result; thus they perform an implicit pop.

Arithmetic and logical operations all use postfix notation. First the operands are pushed, then the operation takes place. Thus %p1 %p2 %p3 %+ %* leaves ***x*** * (***y*** + ***z***) on the stack, where ***x***, ***y***, and ***z*** are parameters 1, 2 and 3. This mechanism is identical to that used by *terminfo*, so its commands can be used freely.

The simpler *termcap* commands do not use a stack mechanism. To support them, JAM uses an automatic parameter sequencing scheme. A current index into the parameter list is maintained. Whenever a parameter is needed on the stack, the current parameter is pushed and the index is incremented. In particular, if an output command is encountered and there is nothing on the stack to output, an automatic push is performed using the current index. The commands `%d %d` output two decimals; the sequence `%p1 %d %p2 %d` does the same thing.

The effect of this scheme is that *termcap* style commands are automatically translated into *terminfo* style. Most of the examples in this document give both styles. Although it is possible to use automatic sequencing and explicit parameter pushes in the same sequence, this practice is strongly discouraged. Explicit pushes of constants with automatic parameter sequencing, however, is a useful combination, as will be seen.

5.3.3

Stack Manipulation and Arithmetic Commands

Commands are available to push parameters and constants. Only four levels of stack are supported, and anything pushed off the end is discarded without warning.

`%p2`	Push the second parameter
`%p11`	Push parameter `11`
`%'x'`	Push the character `x`
`%{12}`	Push the number `12`
`%{0}`	Push binary `0`
`%'0'`	Push ASCII `0`

Various arithmetic and logical operations are supported. They require one or two operands on the stack. If necessary an automatic push will be generated, using the next parameter.

`%'@' %| %| %| %c`	OR 3 parameters with `@`, then output result
`%'@' %| %| %| %c`	Bitwise OR 3 parameters with `@`, then output result

`%'@' %p1 %| %p2 %| %p3 %| %c` same as above

The automatic parameter sequencing mechanism works well in the above example. Since bitwise OR requires two parameters and there is only one on the stack, a push is performed. Note that no push is required to process `%c` since an entry already exists on the stack. Thus only three parameters are consumed and the result of the bitwise OR is output.

`%'SP' %+ %c`	Output the parameter added to the value of a space. See the next section for an alternate.
`%p1 %'SP' %+ %c`	Same as above

The example above first pushes the first parameter, then pushes a space character (`0x20`). The `%+` command adds these values and puts the answer on the stack. `%c` then pops this value and transmits it to the terminal.

5.3.4

Parameter Sequencing Commands

With automatic sequencing of parameters, it is occasionally necessary to skip a parameter. The `%u` command uses up one parameter, by incrementing the parameter index. The `%b` command backs up, by decrementing the parameter index. Both can be given with counts, as `%2u`.

`%d %b %d`	Output the same parameter twice
`%p1 %d %p1 %d`	Same as above
`%p2 %d %p1 %d`	Output in reverse order
`%u %d %2b %d`	Same as above

5.3.5

Output Commands

Because the percent sign is a special character, it must be doubled to output a percent sign. `%c` and `%.` output a character, like `printf`; the latter is supplied for *termcap* compatibility. `%d` outputs a decimal. It has variations that allow for specifying the number of digits, and whether blank–fill or zero–fill is to be used.

`%#z` outputs the specified number of NUL characters (binary zero). It is usually used for padding, to insert a time delay for commands such as erase screen.

`%%`	Output a percent sign
`%d`	Output a decimal, any number of digits, no fill
`%3d`	Output at most 3 digits with blank fill
`%03d`	Output at most 3 digits with zero fill
`%100z`	Send 100 pad bytes to the terminal

`%S(` ***string*** `%)` issues a system command; the string following `%S` is passed to the command interpreter for execution. Since `vid2bin` strips spaces, this text should usually be enclosed in quotes.

`%S('stty tabs'%)]`	System call: `stty tabs`
`%S(stty SP tabs%)`	System call: `stty tabs`
`%S(stty tabs %)`	Mistaken version of above
`%S('keyset \'\''%)`	System call: `keyset ''`
`%S('keyset '''%)`	Mistaken version of above.

`%#w` waits (sleeps) the specified # of seconds. It is not supported on systems where the sleep library routine is unavailable. It is often used as a time delay for `INIT` and `RESET` sequences.

`%2w`	Sleep 2 seconds

Because *termcap* and *terminfo* are inconsistent, `%+` is implemented in two ways. As described in the section above, `%+` can be used to add two operands on the stack and leave the sum on the stack. If the stack has only one entry, an automatic push is generated. However, a special case occurs if the stack is empty—the character following `%+` is added to the next parameter, the sum is output as a character, and the parameter index is incremented. This usage occurs often in *termcap* cursor positioning sequences.

`%+SP`	Output parameter added to the value of space
`%'SP' %+ %c`	Same as above
`%'SP' %p1 %+ %c`	Same as above

5.3.6

Parameter Changing Commands

`%i` increments the next two parameters. It is used almost exclusively in *termcap* cursor positioning sequences. The parameters passed are line and column, with the upper left being (0,0). Many terminals expect the line and column to be relative to (1,1); `%i` is used to increment the parameters. Note that no output is performed, and no parameters are consumed.

`%r` reverses the next two parameters. It is unnecessary if explicit parameter pushes are used; in fact, it should be avoided in that case since the numbering of the parameters will be reversed. This command is often used in cursor positioning sequences where the terminal requires that column be given first and then the line (the default being the other way around).

`ESC [ %i %d ; %d H`	Add 1 to each parameter and send out as decimals
`FS G %r %c %c`	Output column first, then line
`FS G %p2 %c %p1 %c`	Same as above

5.3.7

Control Flow Commands

The general if–then–else clause is `%?` ***expr*** `%t` ***then–part*** `%e` ***else–part*** `%;`. It can be abbreviated by omitting the *if*, thus: ***expr*** `%t` ***then–part*** `%e` ***else–part*** `%;`. The expression ***expr*** is any sequence, including the empty sequence. `%t` pops a value from the stack and tests it, executing ***then–part*** if it is true (non–zero) and ***else–part*** otherwise. ***then–part*** and ***else–part*** may be any sequence, including the empty sequence. If ***else–part*** is empty, %e may be omitted as well; but `%t` is always required, even if ***then–part*** is empty.

If `%t` finds that the stack is empty, it will generate an automatic push of the next parameter as usual. `%t` consumes one parameter; however, the incrementing of the parameter index is delayed until after the entire conditional has been executed. A conditional always consumes exactly one parameter, regardless of which branch is taken or of the content of ***then–part*** or ***else–part***. If either of those use automatic parameter sequencing, they use a local index. Thus even if they consume, say, two parameters, at the end of the conditional the parameter index is reset. When the next command is reached, only one parameter has been consumed.

In each of the following examples, one parameter is consumed, even in the last one where no parameter is output.

`%t ; %c %;`	Output ; AND a character if the parameter is non–zero, otherwise skip the parameter.
`%p1 %t ; %p1 %c %;`	Same
`%? %p1 %t ; %p1 %c %;`	Same
`%? %p1 %t ; %c %;`	Same
`%t ; 5 %;`	Output ; AND 5 if the parameter is non–zero.

In the following two examples, the constant (binary) 1 is pushed, the parameter is compared with 1, and the boolean value is left on the stack. If the value is true, nothing is done; otherwise the parameter is output as a decimal.

```
%? %{1} %p1 %= %t %e %p1 %d %;
%{1} %= %t %e %d %;
```

The following sequence exhibits a simple "either–or" condition that is sometimes used to toggle an attribute on or off. It outputs

ESC (	if the parameter is non–zero, and
ESC)	otherwise:

```
ESC %t ( %e ) %;
```

The ***then–part*** and ***else–part*** may themselves contain conditionals, so else–if can be implemented. This practice is not recommended as it can produce undecipherable sequences. Also, because of the way automatic parameter sequencing is done, the results might be unexpected. It is provided only for *terminfo* compatibility. The list command, described in the next section, is an alternative.

The repeat command is used to perform the same action for several parameters. It is designed to be used with automatic parameter sequencing, and is almost useless if explicit parameter pushes are used. The count is specified after the percent sign. All the commands between `%#` (and `%`) are executed # times.

`%3( %d %)`	Output 3 decimals
`%p1 %d %p2 %d %p3 %d`	Same as previous
`%3( %t %d %; %)`	Output whichever of the first three parameters are non–zero.
`%p1 %t %p1 %d %; %p2 %t %d %; %p3 %t %p3 %d %;`	Same as above
`ESC 0 %9( %t ; %c %; %) m`	Usual ANSI sequence for SGR.
`ESC 0 %? %p1 %t ; 7 %; %? %p2 %t ; 2 ...`	Same as above, assuming that parameter 1 is 7 and parameter 2 is 2.

5.3.8

The List Command

The list command is needed very rarely, but is available as an alternative to a complicated if–then–else–if construction. It implements a simple "select" or "case" conditional. The general format is

`%l(`***value1***`: `***expr1*** `%; `***value2***`: `***expr2*** `%; ... %)`

The values are single character constants representing the various cases. The expression is executed if the value matches the top of stack. At most one expression is executed, (i.e., each case contains a "break"). If the value is missing, the expression is evaluated as a default. For correct operation, the default case must occur last in the list. Note that the colons do not have a leading percent sign, so no selector may be a colon. The `%;` after the last element of the list is not required.

The parameter on the stack (automatically pushed, if necessary) is popped and tested against the various cases. The parameter index is incremented by one after the entire list is processed, even if the expressions use parameters. The following examples are a bit contrived; see the section on color for other examples.

`%l( 0:%; 1:ESC%; :FS %)`	output nothing if the parameter is '0'; ESC if it is '1'; FS otherwise.
`%'0' %= %t %e %'1' %= %t ESC %e FS %; %;`	same result, using "else–if"
`%l( 1:2%; 2:1%; %)`	output '1' if the parameter is '2', '2' if the parameter is '1'; otherwise do nothing

5.3.9

Padding

Certain terminals (or *tty* drivers) require extra time to execute instructions. Sometimes the terminal manual specifies the delay required for each command, but more often than not it is silent on the subject. If random characters appear on the screen, particularly characters that are part of command sequences, time delays may be required.

Delays can be introduced in two ways. `%#w` will cause a wait (sleep) for the specified number of seconds; `%#z` will output the specified number of zeros. The wait command is usually only required in terminal initialization or reset sequences. A hard reset of a terminal sometimes requires a sleep of 1 or 2 seconds. The zero command is occasionally needed on the erase display or erase line commands. Very rarely, the cursor positioning sequence requires padding. The number of zeros to send range from about 5, for very short delays, to several thousand for longer delays. Usually 100 or so is enough for any terminal.

termcap indicates padding by using a number at the beginning of a sequence, which is the number of milliseconds of pad required. *terminfo* uses the syntax `$<#>`. In either case it is easy to convert to the `%#z` notation, using the fact that, at 9600 baud, one character takes one millisecond to output.

`ESC c %2w`	Sleep 2 seconds after terminal reset
`ESC [ J %100z`	100 pad zeros after clear screen
`ESC [ H %1000z`	Long delay of 1000 pad zero

5.4 CONSTRUCTING OR MODIFYING A VIDEO FILE, ENTRY BY ENTRY

5.4.1 Basic Capabilities

LINES indicates the number of lines on the display. The default value is 24. In general one line will be reserved for status and error messages so the maximum form size will usually be one less than the number specified here. (See OMSG, below, for exceptions.) COLMS gives the number of columns on the display. The default value is 80.

`LINES = 25`	24 lines for the screen, 1 for messages
`COLMS = 132`	wide screen

In some windowing environments (e.g., SUN workstations) the values of LINES and COLMS are overridden by the number of lines and columns in the active window.

INIT is a terminal initialization sequence, output by the library function `initcrt`. There is no default; this keyword may be omitted. It is typically used to change the mode of the terminal, to map function keys, select attribute styles, etc. Padding is sometimes required, either with `%#z` or `%#s`.

RESET is a reset–terminal sequence, output by the library function `resetcrt`. There is no default. If given, this keyword should undo the effects of INIT. For many terminals a "hard reset" that resets the terminal to the state stored in non–volatile memory is appropriate.

```
# map 2 function keys, then wait 2 seconds
INIT = %S( "/etc/keyset f1 ^a P ^m" %) \
       %S( "/etc/keyset f2 ^a Q ^m" %) \
       %2w

# load alternate character sets
INIT = ESC)F ESC*| ESC+}

# hard reset, delay, then set tabs
RESET = ESC c %1000z %S('stty tabs'%)
```

On MS–DOS systems only, the INIT and RESET sequences (which are normally not used) may be given a special value to specify the cursor style. With ASCII terminals, escape sequences for setting the cursor style may be included in the INIT and RESET strings in the normal fashion. The format is

```
INIT = C n1, n2, n3
RESET = C n1, n2, n3
```

The first two numbers, ***n1*** and ***n2***, specify the top and bottom scan lines for the cursor block; line 0 is at the top. The optional ***n3*** gives the blink rate, as follows:

1	no cursor
2	fast blink (the default)
3	slow blink
0	fast blink (Not always valid on non–IBM systems)

The standard sequences for a blinking block cursor are:

- For a monochrome monitor: `INIT = C 0, 13, 0`
- For a CGA monitor: `INIT = C 0, 7, 0`
- For a EGA monitor: `INIT = C 0, 13, 2`

If `RESET` is not specified, **JAM** saves and restores the original cursor style.

A scan line is the smallest vertical unit on your display (one pixel wide).

Several additional keywords may be used with `INIT` on MS–DOS systems. These are shown in the table below.

Flag	*Description*
`BIOS`	Specifies that **JAM** should use BIOS calls to do display output rather than writing the video RAM directly.
`WINDOWS`	Relates to Microsoft Windows. Specifies that **JAM** may move the cursor, even if the cursor is invisible.
`XKEY`	Directs **JAM** to use a different BIOS interrupt for keyboard input that recognizes the F11 and F12 keys on an extended keyboard.
`GRAYKEYS`	Distinguishes between gray and white cursor positioning keys. For example, this option allows you to assign one value to the gray up arrow key and another value to the white up arrow key.
`MULTISHIFT`	Permits the use of key sequences using the combinations: `Ctrl-Alt-`, `Shift-Alt-`, and `Ctrl-Shift-Alt-`

`REPT` is a repeat–character sequence. There is no default, since most terminals do not support character repeat. If it is available, it should be given since it can substantially speed up the clearing of windows, painting of borders, etc. This sequence is passed two parameters: the character to be repeated and the number of times to display it. The re-

peat sequence will be used whenever possible, usually for borders and for clearing areas of the screen. If borders do not appear correctly, this sequence may be wrong. A repeat sequence is never used to repeat a control character, and will never extend to more than one line.

REPMAX gives the maximum number of characters REPT can repeat. To check the proper value of REPMAX, first omit it; then in `jxform`, draw a field that extends the entire width of the screen, and hit the TRANSMIT key. If the whole field changes to the underline attribute, REPMAX is not needed. If it fails, experiment by gradually shortening the field to determine the largest possible value of REPMAX.

`REPT= %c ESC F %+?` Output character, then ESC F and the count with 0x3f (the ASCII value of '?') added

`REPMAX= 64` Maximum count for above. Anything over this count will be split into more sequences

`REPT= %p1 %c ESC F %'?' %p2 %+ %c` Same as previous

BOTTOMRT is a simple flag, indicating that the bottom right–hand corner of the display may be written to without causing the display to scroll. If this flag is not present, **JAM** will never write to that position.

BUFSIZ sets the size of the output buffer used by **JAM**. If it is omitted, **JAM** calculates a reasonable default size. You should include this entry only if special circumstances warrant. For example, if you make extensive use of a screen–oriented debugger, you may want to setBUFSIZ to a large value; that effectively disables the delayed–write feature, which may prove troublesome during debugging.

KBD_DELAY assigns a timing interval to keyboard input. Developers who wish to use key sequences that are lead-ins of other sequences must assign a timing interval via KBD_DELAY to determine when a key sequence has ended.

KBD_DELAY may be set to a positive integer between 1 and 10, representing an interval in tenths of a second. Negative integers or 0 represent an interval of indefinitely great length.

5.4.2

Screen and Line Erasure

ED gives the control sequence that erases the display. It is required and must clear all available display attributes, including background color. The correct command can be found in the terminal manual, or in *termcap* as "cl". Some terminals require padding after this command.

`ED = ESC [ J`	Common for ANSI terminals
`ED = CSI J`	ANSI terminals, 8 bit mode
`ED = ESC [ H ESC [ J`	"Home" may be required too
`ED = ESC [ 2 J`	Another variation
`ED = ESC [ 2 J %100z`	With padding
`ED = ^L`	Videotex terminals
`ED = FF`	Same as above

EL gives a sequence that erases characters and attributes from the cursor to the end of the line. If it is not given, **JAM** erases the line by writing blanks. The sequence can be found in *termcap* as `ce`. Padding may be required. `EL =ESC [ K` is common for ANSI terminals; to include padding, use `EL = ESC [ 0 K %100z`.

EW gives a sequence that erases a rectangular region on the screen, to a given background color if available. The only known terminal where this is available is a PC using MS–DOS. Five parameters are passed: start line, start column, number of lines, number of columns, and background color. (If color is not available, the last parameter can be ignored.)

5.4.3

Cursor Position

CUP, absolute cursor position, is required to run **JAM**. This sequence appears in *termcap* as "cm". It takes two parameters: the target line and the target column, in that order and relative to 0. `%i` (increment) can be used to convert them to be relative to 1. ANSI terminals need the line and column as decimals. Other terminals add a fixed value to the line and column to make them printable characters; `%+` is used to implement this. Some typical descriptions follow. All are ANSI standard.

```
CUP = ESC [ %i %d;%d H
CUP = ESC [ %i %d;%d f
CUP = ESC [ %i %p1 %d ; %p2 %d f
CUP = CSI %i %d; %d H
```

Another common scheme is to output the line and column as characters, after adding SP. Examples of coding in the video file follow.

```
CUP = FS C %+SP %+SP
CUP = FS C %'SP' %p1 %+ %c %'SP' %p2 %+ %c
CUP = ESC = %+SP %+SP
```

CUU, CUD, CUF and CUB perform relative cursor movement. CUU moves the cursor up in the same column; CUD moves it down. CUF moves the cursor forward in the same row and CUB moves it back. All take as a parameter the number of lines or columns to move. If sequences exist to move the cursor by only one line or one column, do not specify them.

```
CUU = ESC [ %d A          ANSI cursor up
CUD = ESC [ %d B          Cursor down
CUF = ESC [ % C           Cursor forward
CUB = ESC [ %d D          Cursor back
CUU = CSI %d A            Using 8 bit codes
CUU = ESC [ %{1} %= %t %e %d %; A    Omit the parameter if it is 1
```

The CMFLGS keyword lists several shortcuts **JAM** can use for cursor positioning. They are as follows:

CR	Carriage return (0x0d, or ^M) moves the cursor to the first column of the current line.
LF	Linefeed (0x0a, or ^J) moves the cursor down one line, in the same column.
BS	Backspace (0x08, or ^H) moves the cursor one position to the left, without erasing anything.
AM	Automatic margin: the cursor automatically wraps to column 1 when it reaches the right–hand edge of the display.

Most terminals are capable of the first three. The fourth can frequently be found in *termcap*, as am. It cannot be used on terminals with the *xenl* glitch (i.e., **vt100**–style delayed auto margin.)

5.4.4
Cursor Appearance

CON turns the cursor on in the style desired. Since an underline cursor is difficult to see in an underlined field, we recommend a blinking block cursor. Note that the INIT and RESET sequences can be used to switch between the cursor style used in **JAM** applications and that used on the command line.

COF turns the cursor off. If possible this sequence and CON should be given. Menus (using a block cursor) look better with the regular cursor off. Also **JAM** often must move the cursor around the screen to put text in fields, to scroll arrays, etc. If the cursor is off during these operations, the user is not disturbed by its flickering all over the screen.

Many terminals have no ability to turn the cursor on and off. Although JAM attempts to minimize cursor movement, some flickering is unavoidable.

CON and COF can sometimes be found in the terminal manual as "cursor attributes" and in *termcap* as CO and CF. Here are some examples.

`CON = ESC [`	Cursor on for Videotex terminal
`COF = ESC ]`	Cursor off for Videotex
`CON = ESC [>5l`	Cursor on for some ANSI terminals
`COF = ESC [>5h`	Cursor off for some ANSI terminals
`CON = ESC [?25h`	Another possibility for ANSI terminals
`COF = ESC [?25l`	
`CON = ESC [ 3 ; 0 z`	
`COF = ESC [ 3 ; 4 z`	

The INSON and INSOFF sequences are issued to the terminal when you toggle JAM's data entry mode between insert and overstrike, using the INSERT key. They should change the cursor style, so that you can easily see which mode you are in. On many terminals, changing the cursor style also turns it on; in this case, INSOFF should be the same as COF, or you can omit it altogether. If the cursor style can be changed without turning it on or off, you should give both INSON and INSOFF. INSON and INSOFF use the same escape sequence format as INIT and RESET.

5.4.5

Display Attributes

JAM supports highlight, blink, underline and reverse video attributes. If either highlight or blink is not available, low intensity is supported in its place. One additional attribute keyword is available, called "standout," which can be assigned to any other desired attribute, e.g. dim or italics, if available. The keywords LATCHATT and AREAATT in the video file list the attributes available in each style and associate a character with each attribute.

The set graphics rendition sequences (SGR and ASGR) are each passed twelve parameters. The first nine are the same as used by *terminfo*. The parameters, in order, represent:

1. standout
2. underline
3. reverse video
4. blink

5. dim (low intensity)
6. highlight (bold)
7. blank
8. protect not used, always 0
9. alternate character
10. foreground color (if available)
11. background color (if available)
12. background highlight

If an attribute is desired, the parameter passed is the character associated with the attribute, as explained below. If the attribute is not desired, the parameter passed is (binary) 0. If the video file contains

```
LATCHATT = REVERSE = 7 HILIGHT = 1 BLINK = 5 UNDERLN = 4
```

and a field is to be highlighted and underlined, the SGR sequence will be passed (0, '4', 0, 0, 0, '1', 0, 0, 0). The second and sixth parameters represent underline and highlight; they are set to the corresponding values from LATCHATT. The rest are zero. To make the field reverse video and blinking, SGR would be passed (0, 0, '7', '5', 0, 0, 0, 0, 0) .

If no attributes are specified in the video file, **JAM** will support just two attributes: non–display (done in software anyway) and underline (using the underscore character).

Attribute Types

JAM supports three different kinds of attribute handling. The first is called latch attributes, and is the most common today. The position of the cursor is irrelevant when the attribute setting sequence is sent. Any characters written thereafter take on that attribute. Attributes require no space on the screen. ANSI terminals use this method.

The second is called area attributes. The cursor position is very important at the time the attribute sequence is sent to the terminal. Indeed, all characters from the cursor position to the next attribute (or end of line or end of screen) immediately take on that attribute. Attributes do not occupy a screen position (they are "non–embedded" or "no space"). In this style, **JAM** will position the cursor to the end of the area to be changed, set the ending attribute, then position the cursor to the beginning of the area and set its attribute.

The third is called onscreen attributes. They act like area attributes, but occupy a screen position. (They are "embedded" or "spacing".) This style of attribute handling imposes the condition on the screen designer that fields and/or display areas cannot be adjacent, since a space must be reserved for the attribute. Display of windows may be hampered by lack of space for attributes.

A terminal may have several modes which can be set by the user. It is quite common for a terminal to support both area and onscreen attributes. If so, you should select area

("non–embedded" or "no space") rather than onscreen ("embedded" or "spacing"). Some terminals support one latch attribute and several area attributes simultaneously.

If a terminal has only one attribute style available, it is often user selectable. We recommend that reverse video be selected, since it is attractive in borders. **JAM** supports non–display in software, so that attribute need not be available. Underlines will be simulated (by writing an underscore character) if that attribute is not available.

Usually attribute information is available only in the terminal manual. However, some clues can be found in the *termcap* database. The codes "so", "ul" and "bl" specify standout (usually reverse video), underline and bold respectively. The codes "se", "ue" and "be" give the sequence to end the attributes. The standard ANSI sequences are

```
so=\E[7m:se=\E[0m:ul=\E[4m:ue=\E[0m:bl=\E[1m:be=\E[0m
```

If you find something like these you can be quite sure that ANSI latch attributes are available. If you find entries `ug#1:sg#1` you can be sure that onscreen attributes are in use.

Specifying Latch Attributes

The `LATCHATT` keyword is followed by a list of attributes equated to their associated character. The possible attributes are:

`REVERSE`	Reverse (or inverse) video
`STANDOUT`	User selected standout mode
`BLINK`	Blink or other standout
`UNDERLN`	Underline
`HILIGHT`	Highlight (bold)
`DIM`	Dim (low intensity)
`BLANK`	Non–display (foreground not shown)
`ACS`	Alternate character set (line drawing graphics)
`B_HILIGHT`	Background highlight

The format is `LATCHATT` = ***attribute*** = ***value attribute*** = ***value*** etc. If the equal sign and value are missing, the attribute is given the value (binary) 1.

Most ANSI terminals use latch attributes, and the codes are fairly standardized. The only question is which attributes are supported and how attributes can be combined, if at all. Some ANSI terminals support color, either foreground only or foreground and background. The sequences for color are far less standard.

Terminal manuals often describe the sequence as "set graphics rendition." A common description reads:

```
ESC [ p1 ; p2 ; ... m
```

where p*n* = 0 for normal

1 for bold

5 for blink

Thus ESC [0 m is normal, ESC [1 m is bold, ESC [1 ;5 m is bold and blinking. Often setting an attribute does not "erase" others, so it is best to reset to normal first, using ESC [0; 1 m for bold, ESC[0;1;5m for blinking bold, etc. The coding in the video file is as follows:

```
LATCHATT = HILIGHT = 1 BLINK = 5 UNDERLN = 4 REVERSE = 7
SGR = ESC [ 0 %9(%t ; %c %; %) m
```

The meaning of the above SGR sequence is as follows. The sequence is passed 11 parameters, each 0 (if the attribute is not to be set) or the character in the LATCHATT list. First, ESC [0 is output. The %t test, repeated 9 times, causes the zero parameters to be skipped. A non–zero parameter causes a semicolon and the parameter to be output. Finally, the character m is output. If normal attribute is wanted, all parameters will be 0, and the output sequence will be ESC [0 m. If only underline is wanted, it will be ESC [0 ; 4 m. If highlighted, blinking, and reverse video are desired, the output will be

```
ESC [ 0; 7 ; 5 ;1 m.
```

Some terminals (or emulators) will not accept the method of combining attributes used above. In that case, one sequence followed by the next might work, e.g. ESC [1 m ESC [7m. Some terminals cannot combine attributes at all. Here are some more ANSI and near–ANSI examples:

```
LATCHATT= HILIGHT=1 BLINK=5 UNDERLN=4 REVERSE=7
```
"standard" ANSI terminal

```
LATCHATT= DIM=2 REVERSE=7 UNDERLN=4 BLINK=5
```
ANSI with low intensity but no highlight

```
LATCHATT= REVERSE=7
```
only one attribute available

```
SGR = ESC [ 0 %9(%t ; %c %; %) m
```
repeat of previous example

```
SGR = ESC [ 0 m %9(%t  ESC [ %c m %; %)
```
attributes cannot be combined

```
SGR = %u ESC [ 0 %5(%t ; %c %; %) m
```
skip parameters that are always 0

In the next LATCHATT/SGR example we will use explicit pushes to select the appropriate parameter. The second pair is the same as the first, but the attribute is treated as a boolean. The first uses the optional %?, the second omits it.

```
LATCHATT = DIM = 2SGR = ESC [ m %? %p5 %t ESC [ 2 m %;

LATCHATT = DIMSGR = ESC [ m %t ESC [ 2 m %;
```

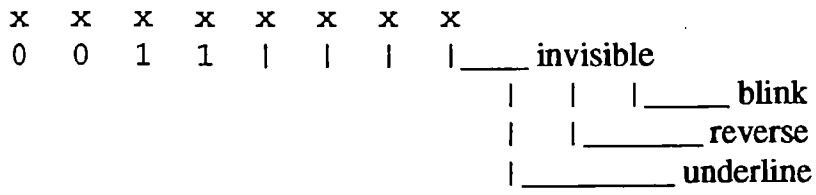
The following is suitable for terminals that support all attributes but cannot combine them. It selects one attribute giving preference to REVERSE, UNDERLN, BLINK and HILIGHT in that order. It uses a complicated "if–then–elseif–elseif–elseif" structure. Automatic parameter sequencing cannot be relied on, so explicit parameter pushes are used.

```
LATCHATT = HILIGHT BLINK UNDERLN REVERSE
SGR = ESC [ %p3 %t 7 %e %p2 %t 4 %e %p4 %t 5 %e\
    %p6 %t 1 %; %; %; %; m
```

Some terminals use bit–mapped attributes. Terminal manuals are not usually explicit on this. Often they use tables like the following:

n	*Visual attribute*	*n*	*Visual attribute*
0	normal	8	underline
1	invisible	9	invisible underline
2	blink	:	underline and blink
3	invisible blink	;	invisible underline and blink
4	reverse video	<	reverse and underline
5	invisible reverse	=	invisible reverse and underline
6	reverse and blink	>	reverse, underline and blink
7	invisible reverse and blink	?	invisible reverse, underline and blink

After poring over the ASCII table for a while, it becomes clear that this is bit–mapped, with the four high–order bits constant (0x30) and the four low–order bits varying, like this:



This can be coded in the video file as follows. The attributes are OR–ed with a starting value of '0' (0x30).

```
LATCHATT = BLINK = 2 REVERSE = 4 UNDERLN = 8
SGR = ESC G %'0' %9( %| %) %c
```

The following gives an example for use with a Videotex terminal. All are equivalent: the bits are OR–ed together with a starting value of 0x40, or @, and the result is output as a character.

```
LATCHATT= UNDERLN=DLE BLINK=STX REVERSE=EOT HILIGHT=SP
LATCHATT= UNDERLN= ^P BLINK= ^B REVERSE= ^D HILIGHT= SP
LATCHATT= UNDERLN= 0x10 BLINK= 0x02 REVERSE= 0x04 \
     HILIGHT= 0x20

SGR= FS G %u %'@' %5( %| %) %c

LATCHATT= UNDERLN= P BLINK= B REVERSE= D HILIGHT= '
SGR = FS G %'@' %9( %| %) %c
```

Some terminals that use area attributes will support a single latch attribute. It is often called "protected" and is used to indicate protected areas when the terminal is operated in block mode. The following example switches between protected and unprotected modes in order to use low intensity. (Be aware that a terminal might become very slow when using the protect feature.) The SGR sequence depends only on the attribute being non–zero, so no value is necessary:

```
LATCHATT = DIM
SGR = ESC %?%t ) %e ( %;
```

Specifying Area Attributes

Area or onscreen attributes are specified like latch attributes. The AREAATT keyword lists the area or onscreen attributes that are available and associates a character with each. As for latch attributes, REVERSE, BLINK, STANDOUT, ACS, UNDERLN, HILIGHT and DIM are available. In addition, several flags are available to specify how the attributes are implemented by the terminal. The flags are:

ONSCREEN	The attribute uses a screen position
LINEWRAP	The attribute wraps from line to line
SCREENWRAP	The attribute wraps from bottom of screen to top
REWRITE	Must rewrite attribute when writing character

Area and onscreen attributes modify all characters from the start attribute to the next attribute or to an end, whichever is closer. If there is no end, use SCREENWRAP. If the end is the end of screen, use LINEWRAP. If end is the end of the line, omit both wrap flags. Some terminals allow the user to select the style. For onscreen attributes, screen

wrap is best and line wrap a good second best; for area attributes the choices are about the same. If the attribute takes up a screen position, use the ONSCREEN flag.

```
AREAATT = REVERSE = i UNDERLN = _ BLINK = b DIM =l
ASGR = ESC s r %u %5(ESC s %c %)
```

```
AREAATT= BLINK= 2 DIM= p REVERSE= 4 UNDERLN= 8 \
    ONSCREEN LINEWRAP
ASGR = ESC G %u %'0' %5( %| %) %c
```

On some terminals writing a character at the position where an attribute was set can remove the attribute. Immediately after placing the attribute the character may be written with no problems; however, the next time a character is written there, the attribute will disappear. In this case, use the REWRITE flag to tell **JAM** to reset the attribute before writing to that position. The following example is for the Televideo 925:

```
AREAATT = REVERSE = 4 UNDERLN = 8 BLINK = 2 REWRITE
ASGR = ESC G %'0' %9( %| %) %c
```

Yet other terminals restrict the number of attributes that are available on a given line. If possible, give a "remove attribute" sequence, ARGR. Changing an attribute to normal is not the same as removing it; a normal attribute will stop the propagation of a previous attribute, but a removed attribute will not. If the maximum number of attributes is small, **JAM**'s performance may be limited. ARGR is desirable because having many attributes onscreen can dramatically slow performance, since **JAM** must keep rewriting them as attributes change.

If there is a remove attribute sequence, **JAM** will use it to remove repeated attributes, to avoid exceeding the maximum number of attributes on a line. If there is no maximum, the remove attribute sequence can be omitted. Indeed, it often makes the screen "wiggle," which is very unpleasant for the viewer.

```
AREATT = REVERSE = Q UNDERLN = '
ASGR = ESC d %u %'@' %5( %| %) %c
ARGR = ESC e
```

Attributes that Do Not Affect Space

A list of attributes which would not change the appearance of a character cell containing a space may be given. For example,

```
SPXATT = BOLD DIM BLANK BLINK COLOR
```

For efficiency, this is used to reduce the number of characters sent to a screen. It defaults to COLOR BLANK HILIGHT DIM. It may be turned off entirely by

```
SPXATT =
```

Color

JAM supports eight foreground and background colors. The COLOR keyword is used to associate a character with each color, just as LATCHATT associates a character with

each attribute. The CTYPE entry has flags that tell **JAM** that background color is available. Only the three primary colors need be specified in the video file. If the other colors are not there, they will be generated according to the following rule:

BLACK =	BLUE & GREEN & RED
BLUE	Must be specified
GREEN	Must be specified
CYAN =	BLUE | GREEN
RED	Must be specified
MAGENTA =	RED | BLUE
YELLOW =	RED | GREEN
WHITE =	RED | GREEN | BLUE

The tenth parameter to SGR or ASGR is the character representing the foreground color; the eleventh is that representing the background color (it is 0 if background color is not available). Many ANSI terminals set foreground color with the sequence ESC [3*x* m, where *x* ranges from 0 for black to 7 for white. Background color is often set with ESC [4*x* m. The order of the colors varies from terminal to terminal.

On color terminals, REVERSE often means black on white. If background color is available, **JAM** can do better if REVERSE is not specified in the video file. It will use the specified color as the background, and either black or white as the foreground. The following is often suitable for a color ANSI terminal:

```
LATCHATT = HILIGHT = 1 BLINK = 5
COLOR = RED = 4 GREEN = 2 BLUE = 1 BACKGRND
SGR = %3u ESC [ 0 %3( %?%t ; %c %; %) ; %3u 3%c ; 4%c m
```

or

```
SGR = %3u ESC [ 0 %5( %?%t ; %c %; %) m ESC [ 3%c;4%c m
```

or

```
LATCHATT = HILIGHT BLINK
SGR = ESC [ 0 %?%p4%t ;5 %; %?%p6%t ;1 %; m \
    ESC [ 3%p10%c; 4%p11%c m
```

If the terminal has a unique sequence for each color, a list command works well. In the following example, the ANSI attribute sequence (ESC [0 ; p1 ; p2 ; ... m) is used and the values for the colors are:

```
cyan    >1
magenta 5
blue    5 ; > 1
yellow  4
green   4 ; > 1
red 4 ; 5
black   4 ; 5 ; > 1
```

```
LATCHATT = REVERSE = 7 HILIGHT = 2

COLOR = CYAN = 0 MAGENTA = 1 BLUE = 2 YELLOW =3 \
    GREEN = 4 RED = 5 BLACK = 6 WHITE = 7

SGR = ESC [ 0 %p3%t;7%; %p6%t;2%; \
    %l( 0:;>1%; 1:;5%; 2:;5;>1%; 3:;4%; \
    4:;4;>1%; 5:;4;5%; 6:;4;5;>1 %) m
```

Some terminals use ESC [2 ; ***x*** ; ***y*** m to set color and other attributes. Here ***x*** is the foreground color and ***y*** is the background color; both numbers range from 0 to 7. If highlight is desired in the foreground, 8 should be added to ***x***. If blink is desired, 8 should be added to ***y***. The following video entries satisfy these requirements:

```
LATCHATT = HILIGHT = 8 BLINK = 8
COLOR = RED = 4 GREEN = 2 BLUE = 1 BACKGRND
SGR = ESC [ 2 ; %p10 %p6 %+ %d ; %p11 %p4 %+ %d m
```

5.4.6

Message Line

JAM usually steals a line from the screen to display status text and error messages. Thus a 25–line screen (as specified in the LINES keyword) will have 24 lines for the form itself, and one for messages. This use of a normal screen line for messages is the default. Some terminals have a special message line that cannot be addressed by normal cursor positioning. In that case, the OMSG sequence is used to "open" the message line, and CMSG to close it. All text between these sequences appears on the message line. No assumption is made about clearing the line; **JAM** always writes blanks to the end of the line.

```
OMSG = ESC f
CMSG = CR ESC g
```

If the OMSG line keyword is present, **JAM** uses all the lines specified in the LINES keyword for forms.

Terminals that use a separate message line may use different attributes on the status line than on the screen itself. **JAM** provides some support for this circumstance; for very complicated status lines, the programmer must write a special routine and install it with the statfnc call. (See the *Programmer's Guide* for details.) The keyword MSGATT lists the attributes available on the message line. This keyword takes a list of flags:

REVERSE	Reverse video available
BLINK	Blink available
UNDERLN	Underline available
HILIGHT	Highlight (bold) available

DIM	Dim (low intensity) available
STANDOUT	User defined standout mode
ACS	Alternate character set
LATCHATT	All latch attributes can be used
AREAATT	All area attributes can be used
NONE	No attributes on message line
ONSCREEN	Area attributes take a screen position

The attribute for the message line must have been specified as either a latch or area attribute, and the sequence to set it must be given in the SGR or ASGR keyword. For example, if REVERSE is listed in MSGATT and REVERSE is a latch attribute, then SGR is used to set it. Attributes that appear in MSGATT and don't appear in either LATCHATT or AREAATT are ignored.

JAM must determine the correct count of the length of the line. Thus it is important to know whether area attributes are onscreen or not. It is not uncommon for area attributes to be non–embedded on the screen but embedded on the status line. The keyword ONSCREEN may be included in MSGATT to inform **JAM** of this condition.

```
LATCHATT = DIM
AREAATT = REVERSE UNDERLN BLINK
MSGATT = REVERSE UNDERLN BLINK ONSCREEN
MSGATT = AREAATT ONSCREEN
```

The two MSGATT entries are equivalent. They show a case where only area attributes are available on the message line and they take a screen position. The area attributes in the normal screen area do not.

5.4.7
Soft Key Labels

If you are using soft keys in an application, you must have a KPAR entry in the video file to display the soft key labels.[5]

Certain terminals set aside areas on the screen, typically two lines high and several characters wide, into which descriptive labels for the terminal's function keys may be written. If your terminal does not provide this hardware support, **JAM** can simulate soft keys.

The KPAR entry gives the number and width of the function key labels, and has the form

5. The source `jmain.c` and `jxmain.c` must be modified to use soft keys. See the *Programmer's Guide* for directions. See the *Author's Guide* for information on creating and using keysets. If you write your own routines for handling soft keys, then you do not need to alter the main routine, but you will need a KPAR entry.

```
KPAR = NUMBER = number of labels LENGTH = width of area.
```

It is passed the following parameters:

- NUMBER= Specify the number of labels in a keyset row. Required.
- LENGTH= Specify the length of each label. Required. If NUMBER * (LENGTH + 1) is greater than the number of columns in the screen, not all the labels will be displayed.
- ATTRIBUTE Indicates that labels may utilize the display attributes of the terminal. Optional.
- SIMULATE On terminals which do not provide hardware support for soft keys, include SIMULATE in the KPAR entry. This steals two lines from the screen and simulates soft keys.
- 1LINE This parameter is used with SIMULATE. Soft keys are simulated, but only one terminal line is used instead of two. Optional.

If your terminal provides hardware support for soft keys, use the KSET entry to specify the character sequence for writing text into a label area. If keysets are simulated, this entry is ignored. KSET is passed the following parameters:

- The number of the label (a number between one and the value given NUMBER in the KPAR entry)
- Pointer to the first label line. Points to a null–terminated string of length of LENGTH.
- Pointer to the second label line. Points to a null–terminated string of length of LENGTH.
- Attribute parameters. There may be up to 12 attribute parameters.

See your terminal's documentation for the specific sequence.

Here is an example of the entries for the HP–2392A, a terminal which provides hardware support for soft keys:

```
KPAR = NUMBER = 8 LENGTH = 8
KSET = ESC & f %d k 16 d 0 L %s ESC & j B
```

Here is an example for a PC (no hardware support):

```
KPAR = NUMBER = 8 LENGTH = 8 SIMULATE ATTRIBUTE
```

5.4.8

Graphics and International Character Support

JAM has support for eight–bit ASCII codes as well as any graphics that the terminal can support in text mode. Bit–mapped graphics are not supported. Just as the key translation tables give a mapping from character sequences to internal numbers, the GRAPH table in the video file maps internal numbers to output sequences. The only character value that may not be sent is 0.

Some terminals have a special "compose" key, active in eight–bit mode. Generally, you would press the compose key followed by one or two more keys, generating a character in the range `0xa0` to `0xff`. **JAM** can process such characters as normal display characters, with no special treatment in the video file.

Other terminals have special keys that produce sequences representing special characters. The modkey utility can be used to map such sequences to single values in the range `0xa0` to `0xfe`. (See the *Programmer's Guide* for a way to use values outside that range.)The video file would then specify how these values are output to the terminal.

Often, to display graphics characters, a terminal must be told to "shift" to an alternate character set (in reality, to address a different character ROM). The video file's GRAPH table tells which alternate set to use for each graphics character, and how to shift to it. Whenever **JAM** is required to display a character, it looks in the GRAPH table for that character. If it is not there, the character is sent to the terminal unchanged. The following section describes what happens if it is in the table.

Graphics Characters

JAM supports up to three alternate character sets. The sequences that switch among character sets are listed below. Modes 0 through 3 are locking shifts. All characters following will be shifted, until a different shift sequence is sent. Modes 4 through 6 are non–locking or single shifts, which apply only to the next character. You may need to use the INIT entry to load the character sets you want for access by the mode changes.

MODE0	switch to standard character set
MODE1	alternate set 1
MODE2	alternate set 2
MODE3	alternate set 3
MODE4	...
MODE5	
MODE6	

Different modes can be used to support foreign characters, currency symbols, graphics, etc. JAM makes no assumption as to whether the mode changing sequences latch to the alternate character set or not. To output a character in alternate set 2, JAM first outputs the sequence defined by MODE2, then a character, and finally the sequence defined by MODE0 (which may be empty, if the others are all non–locking). Here are three examples; the second one is ANSI standard.

```
MODE0 = SI
MODE1 = SO
MODE2 = ESC n
MODE3 = ESC o

MODE0 = ESC [ 10 m
MODE1 = ESC [ 11 m
MODE2 = ESC [ 12 m
MODE3 = ESC [ 13 m

MODE0 =
MODE1 = SS1
MODE2 = SS2
```

Any of the MODE*n* strings may also contain a list of attributes. When a character in that mode is displayed, that attribute will be added to whatever attribute is already in effect. On some terminals, like the HP, only an attribute is required. For example,

```
MODE4 = ACS
```

which would force all mode 4 characters to be displayed using the alternate character set.

Any character in the range 0x01 to 0xff can be mapped to an alternate character set by use of the keyword GRAPH. The value of GRAPH is a list of equations. The left side of each equation is the character to be mapped; the right side is the number of the character set (0, 1, 2, 3), followed by the character to be output. Any character not so mapped is output as itself. For example, suppose that 0x90 =1 d appears in the GRAPH list. First the sequence listed for MODE1 will be sent, then the letter d, and then the sequence listed for MODE0.

In the following example, 0x81 is output as SO / SI, 0xb2 as SO 2 SI, and 0x82 as ESC o a SI. LF, BEL and CR are output as a space, and all other characters are output without change. This output processing applies to all data coming from JAM. No translation is made for direct calls to printf, putchar, etc. Thus \n and \r will still work correctly in printf, and putchar (BEL) still rings the terminal bell.

```
MODE0 = SI

MODE1 = SO

MODE2 = ESC n

MODE3 = ESC o
```

```
GRAPH = 0x81 = 1 / 0xb2 = 1 2 0x82 = 3 a LF = 0 SP\
    BEL = 0 SP CR = 0 SP
```

For efficiency, we suggest that you use single shifts to obtain accented letters, currency symbols, and other characters that appear mixed in with unshifted characters. Graphics characters, especially for borders, are good candidates for a locking shift.

It is possible, though not recommended, to map the usual display characters to alternates. For example, GRAPH = y = 0z will cause the y key to display as z. Graphics characters are non–portable across different displays, unless care is taken to ensure that the same characters are used on the left–hand side for similar graphics, and only for a common subset of the different graphics available.

The GRTYPE keyword provides a convenient shortcut for certain common graphics sets, each denoted by another keyword. The format is GRTYPE = ***.type*** An entry in the GRAPH table is made for each character in the indicated range, with mode 0. If the mode is not 0, you must construct the GRAPH table by hand. The GRTYPE keywords are:

ALL	0xa0 through 0xfe
EXTENDED	same as ALL.
PC	0x01 through 0x1f and 0x80 through 0xff
CONTROL	0x01 through 0x1f, and 0x7f; same as CONTROL
C1	0x80 through 0x9f, plus 0xff

The GRTYPE keywords may be combined.

5.4.9

Borders and Line Drawing

The characters constituting the border and line drawing styles may be specified in the video file.

Borders

Ten different border styles may be selected when a screen is designed. They are numbered 0 to 9, with style 0 being the default (and the one all **JAM** internal forms use). It is usually reverse video spaces, but is replaced by Is if reverse video is not available. Border styles may be specified in the video file. Otherwise, the following defaults are used:

0.
```
IIIII
I   I
IIIII
```

1.
```
_____
|   |
_____
```

2.
```
+++++
+   +
+++++
```

3.
```
=====
|   |
=====
```

4.
```
%%%%%
%   %
%%%%%
```

5.
```
.....
:   :
.....
```

6.
```
*****
*   *
*****
```

7.
```
\\\\\
\   \
\\\\\
```

8.
```
/////
/   /
/////
```

9.
```
#####
#   #
#####
```

The keyword BORDER specifies alternate borders. If fewer than 10 are given, the default borders are used to complete the set. The data for BORDER is a list of 8 characters per border, in the order: upper left corner, top, upper right corner, left side, right side, lower left corner, bottom, lower right corner. The default border set is:

```
BORDER =   SP  SP  SP  SP  SP  SP  SP  SP \
    SP  _  SP  |   |   |   _   |  \
    +   +  +   +   +   +   +   +  \
    SP  =  SP  |   |   SP  =   SP \
    %   %  %   %   %   %   %   %  \
    .   .  .   :   :   :   .   :  \
    *   *  *   *   *   *   *   *  \
    \   \  \   \   \   \   \   \  \
    /   /  /   /   /   /   /   /  \
    #   #  #   #   #   #   #   #
```

Another example, using the PC graphics character set:

```
BORDER =   SP  SP  SP  SP  SP  SP  SP  SP   \
    0xda   0xc4   0xbf   0xb3   0xb3   0xc0   0xc4   0xd9 \
    0xc9   0xcd   0xbb   0xba   0xba   0xc8   0xcd   0xbc \
    0xd5   0xcd   0xb8   0xb3   0xb3   0xd4   0xcd   0xbe \
    0xd6   0xc4   0xb7   0xba   0xba   0xd3   0xc4   0xbd \
```

```
0xdc    0xdc    0xdc    0xdd    0xde    0xdf    0xdf    0xdf \
.   .   .   :   :   :   .   .     \
0xb0    0xb0    0xb0    0xb0    0xb0    0xb0    0xb0    0xb0 \
0xb2    0xb2    0xb2    0xb2    0xb2    0xb2    0xb2    0xb2 \
0xbd    0xbd    0xbd    0xbd    0xbd    0xbd    0xbd    0xbd
```

In the same way as for MODE*n*, attributes may be specified for each set of border characters. For example,

```
BORDER =    SP  SP  ...     SP  REVERSE \
    ;   A   ... +   ACS \
```

If there is a GRAPH entry in the video file, you can use the graphics character set of the terminal for borders. Choose some numbers to represent the various border parts. The GRAPH option can be used to map these numbers to a graphics character set. The numbers chosen are arbitrary, except that they should not conflict with ordinary display characters. Even if the extended 8 bit character set is used, there are unused values in the ranges 0x01 to 0x1f and 0x80to 0x9f.

The keyword BRDATT can be used to limit the attributes available in the border. Normally HILIGHT (or DIM) and REVERSE are used; however, if the terminal uses onscreen attributes or can hold only a few attributes per line, it may be better to prohibit attributes in borders. This is accomplished by BRDATT = NONE.

The flags used in MSGATT can also be used with BRDATT; however, the only attributes available are HILIGHT, DIM, and REVERSE.

Line Drawing

Ten different sets of line draw characters may be specified. These are used by the Line Graphics Style Screen in Authoring. The characters are specified with the BOX keyword, similar to BORDER. BOX is a list of either five or thirteen characters per set. If only five characters are specified the remaining eight are taken from the corresponding BORDER set.

Although the format in the video file is similar, **JAM** uses BOX and BORDER differently. BORDER is portable across different platforms because **JAM** saves a border as its style number in the screen file. **JAM** saves line drawing as display data. For developers creating portable applications, we recommend that you avoid assigning graphic characters to the BOX keyword. Instead, use characters which are displayable on all the terminals.

5.4.10

Indicators

Shifting and Scrolling

Shift/Scroll indicators (ARROWS keyword) are used to indicate the presence of off-screen data in shifting/scrolling fields. The default characters for this purpose are <, >, and X for shifting; ^, v, and X for scrolling. (The character X is used when two shifting/scrolling fields are next to each other; it represents a combination of both < and >.)

These indicators can be changed to any characters desired.

```
ARROWS = . . .

GRAPH = 0x1b = 0 0x1b   0x1a = 0 0x1a   0x1d = 0 0x1d
ARROWS = 0x1b 0x1a 0x1d .......

ARROWS =  < >  X ^ v X

MODE0 = SI
MODE1 = SO
GRAPH = 0x80 = 1a 0x81 = 1x 0x82 = 1&
ARROWS = 0x80 0x81 0x82
```

If the screen background color is white, yellow or cyan then the shift/scroll indicators are black. For all other screen background colors the indicators are white. Indicator colors cannot be changed by the developer.

Bell

The BELL sequence, if present, will be transmitted by the library function `bel` to give a visible alarm. Normally, that routine rings the terminal's bell. Such a sequence can sometimes be found in the *termcap* file under vb.

Selection Box for Groups

If there are no entries for CBSEL and CBDSEL in the video file, the internal defaults are X for CBSEL and a blank for CBDSEL. If a radio button or a checklist has a box edit, these characters are used to indicate which fields are selected and which are not. You may add these entries to the video file to override the defaults. For example,

```
CBSEL   =   y
CBDSEL  =   n
```

As a result, **JAM** will put a y in the box of a selected occurrence. Pressing NL will deselect the occurrence, and **JAM** will put an n in the box.

5.4.11
Drivers

Mouse

If you are using a mouse in a **JAM** application, you must specify the driver in the video file. At this time, mouse support is available for PC users. The following should be added to the PC video file,

```
MOUSEDRIVER=    PC
```

Block Mode

If an application program calls `sm_blkinit` before installing a block mode driver with `sm_install`, **JAM** will look for a `BLKDRVR` entry in the video file. This entry gives the name of a default driver.

```
BLKDRVR =    3270
```

Please see the chapter on block mode in the *Programmer's Guide* for information on using block mode and writing block mode drivers.

5.4.12
Miscellaneous

Display Cursor Position on the Status Line

Use this keyword to display the current cursor position on the status line if desired. When possible, **JAM** uses non–blocking keyboard reads. If no key is obtained within a specified time, the cursor position display is updated. This allows fast typists to type at full speed; when the typist pauses, the cursor position display is updated. The keyword `CURPOS` specifies the time–out delay, in tenths of a second. If the keyword is omitted, or is 0, there will be no cursor position display. Many terminals display the cursor position themselves.

The delay depends on the baud rate and the terminal itself. It should be chosen so that typing is not slowed down. If the terminal has its own display, `CURPOS` should be omitted. If there is no non–blocking read, a non–zero value of `CURPOS` enables the display and zero disables it.

`CURPOS = 1` update display every .1 second (use on fast systems)

`CURPOS = 3` every .3 second (reasonable for most)

`CURPOS = 7` at low baud rates

`CURPOS = 0` no display, same as omitting keyword

Compression

The entry

```
COMPRESS = JTERM
```

implements data compression for *Jterm* users.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

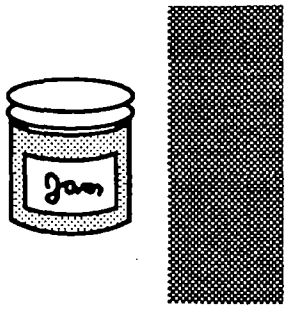
W

X

Z

Utilities Guide

TABLE OF CONTENTS



Chapter 1
Introduction

The `util` directory contains all the **JAM** executables, including `jxform`. The utilities fall into several categories:

Screen and Application Management: The `jxform` utility supplies the Screen, Keyset, and Data Dictionary Editors. This utility is fully documented in the **JAM** *Author's Guide*. The other utilities listed here supplement the capabilities of the Screen Editor.

- `f2asc`

  creates an ASCII description of a **JAM** screen, which may be modified, and converted back to a binary screen file. Useful for storing screens under code control. (Detailed reference on p.34)

- `formlib`

  creates and maintains libraries for screens and JPL procedures. (p. 52)

- `jpl2bin`

  converts JPL files to binary; eliminates run–time compilation of JPL file and permits JPL files to be added to libraries and memory resident lists. (p. 59)

- `jxform`

  invokes the authoring utility. (see **JAM** *Author's Guide)*

- `txt2form`

  creates **JAM** screen from any ASCII text file. (p. 89)

Data Dictionary Management: Using these utilities is an easy way to edit data dictionary files and to edit screen files based on data dictionary entries.

- `dd2asc`

  converts data dictionary from binary to ASCII format; the ASCII file may

be edited and converted back to a binary data dictionary file. (Detailed reference on p. 14)

- `ddmerge`
  combines separate data dictionaries into one data dictionary. (p. 32)
- `ddsort`
  sorts data dictionary entries alphabetically. (p. 33)
- `f2dd`
  creates or updates data dictionary from **JAM** screens. (p. 44)
- `jamcheck`
  compares data dictionary entries with screen fields and groups; permits developer to change fields and groups to match data dictionary entries. (p. 54)

Programming: **JAM** provides utilities to convert binary **JAM** files into programming language files. By default, "C" is the supported language.

- `bin2c`
  converts any **JAM** binary file to C source code. (Detailed reference on p. 11)
- `dd2struct`
  creates data structures from data dictionary entries. (p. 26)
- `f2struct`
  creates data structures from fields and groups. (p. 46)

Documentation: These utilities produce readable ASCII listing which may simplify the documentation and testing of your screens, data dictionaries, and **JAM** applications. The reports created with these utilities cannot be converted back to binary **JAM** files.

- `jammap`
  produces reports which show the relations among screens in a **JAM** application. (Detailed reference on p. 57)
- `lstdd`
  creates ASCII listing of a data dictionary. (p. 63)
- `lstform`
  creates ASCII listing of a screen file. (p. 65)

Configuration: **JAM** depends on several binary files to tell it how to run on a particular computer or terminal. ASCII and binary versions of these files—key translation file, video file, message file, and setup file—are usually stored in the `config` directory. The `util` directory supplies the programs for converting the ASCII configuration files to their binary form. These utilities are

- `key2bin`
  converts ASCII key file to binary. (Detailed reference on p. 61)

- `msg2bin`
  converts ASCII message file to binary. (p. 86)
- `var2bin`
  converts ASCII smvars and setup files to binary. (p. 90)
- `vid2bin`
  converts ASCII video file to binary. (p. 92)

In addition there are utilities to help you create key and video files. They are:

- `modkey`
  invokes editor for creating, modifying, and testing key translation files. (p. 67)
- `term2vid`
  creates simple video file for UNIX systems from termcap or terminfo entries. (p. 88)

Upgrading: For users who are updating from earlier releases, utilities are provided for converting screen and data dictionary files to **JAM** 5 formats.

- `dd3to5, dd4to5`
  upgrades release 3 and 4 data dictionaries to release 5. (Detailed reference on pp. 29, 30)
- `f3to5, f4to5`
  upgrades release 3 and 4 screens to release 5. (p. 49, 50)

Transporting: When transferring files between systems, this utility may be used to convert binary files and store them in a single hexadecimal ASCII file.

- `bin2hex`
  converts binary files to hexadecimal ASCII and vice versa. (Detailed reference on p. 13)

1.1
FEATURES AND OPTIONS COMMON AMONG UTILITIES

The following section describes command–line options and file–handling procedures shared by most or all of the **JAM** configuration utilities. When a utility deviates from this standard, as a few do, the section describing that utility will make it clear.

1.1.1

Getting Help

Command–line options are identified by a leading hyphen. Some systems also support a / to begin command–line options. Except for `modkey`, you can obtain a usage summary by entering the name of the utility followed by `-h`. For example:

```
formlib -h
```

A brief description of the utility's arguments and command options will be displayed. As for `modkey`, once you invoke the utility, you will be presented with a list of options, including help.

Some command line errors, such as missing or improper use of flags, or missing file arguments, will cause a display of the utility's help banner and a brief description of the error.

The help banners use the following symbols:

|	The pipe symbol separates options that may not be used together.
{ }	One option must be chosen from the list enclosed in curly braces. The options are separated by pipe symbols. You must use exactly one option from this list to execute the utility.
[ ]	These options are supplementary. If there are no pipe symbols inside the square brackets, you may use any combination of these options, or use none of them. If the options are separated by pipes, you may use one or none of the options from this list.
< >	These brackets indicate that a file name is a required argument.
[< >]	These brackets indicate that zero or one file name may be used an argument.
...	Ellipsis indicate that additional input files are permitted as arguments.

The help banner descriptions are brief. You should see this document for a more complete description of options, arguments, and defaults for each of the utilities.

1.1.2

Input and Output Files

With a few exceptions, utilities accept multiple input files. If your operating system supports this feature, multiple input files may use a wildcard specification (e.g., `*.jam` for all files with the `.jam` extension). Some utilities combine information from the inputs to create a listing; others perform some transformation on each input individually. The utilities usually will not allow you to overwrite an input file with an identically named output

file. Most utilities will also refuse to overwrite an existing output file; you may force the overwrite with the -f option.

Utilities that generate one output file for each input will, by default, give output files the same name as the corresponding input, but with a different extension. Each utility has its own default extension; in addition, each one supports a -e option that enables you to specify the output file extension. For example:

```
f4to5 -e new mytop.mnu myscreen.win
```

converts the Release 4 screens mytop.mnu and myscreen.w into Release 5 format, and puts the new screens in mytop.new and myscreen.new. The form -e- makes the output file extension null, although not all of the utilities support this option.

Utilities that generate multiple output files, also support an –o option, which directs the output to one named file. For example:

```
lstform -omylist *.frm
```

lists all the screens in the current directory, and places the listing in a file named mylist. You may use a blank space between –o and the file name. For example,

```
f2struct -o screenrecs.h screen1.jam screen2.jam
```

generates C data structures for screen1 and screen2, and places them both in screenrecs.h. Without the -o option, it would have created two output files, screen1.h and screen2.h.

Another form of this option, -o-, sends the program's output to the standard output file rather than to a disk file. msg2bin and dd3to5 support –o, but not –o–. Any other utility that supports –o will support –o–.

If you use both the -e and the -o options, only the -o will be used, and -e will be ignored.

By default, if an input file name contains a path component, a utility will strip it off in generating the output file name; this usually means that output files will be placed in your default directory. You may supply a -p option to have the path left on, that is, to create the output file in the same directory as the input. We strongly discourage the use of –p where it will overwrite input files. In particular, –p should not be used with the conversion upgrade utilities (i.e., dd4to5, f4to5, etc.).

1.1.3
File names and Extensions

JAM runs on many operating systems, each with different rules for file names. You should follow the rules of your operating system when creating and specifying file

names. Any utility that accepts a file name as an argument will also accept a pathname (the full name of the path from the root through the tree of directories to a particular file). Again, follow the rules of your system for specifying pathnames and use the system's path separator character. **JAM** will use a pathname as you supply it, without any alterations.

JAM, like many other software systems, uses extensions to identify the contents of a file. We have tried to make our conventions flexible: extensions are not required, but are supplied by default, and the default can always be overridden. There are three distinct operations involving file extensions:

1. *Finding and modifying files.* In **JAM**, several setup variables control the use and format of extensions. If the setup file configures **JAM** to recognize extensions, then `jxform`, the **JAM** run–time system, and several of the utilities assume that screen files have a common extension, which is specified by the setup variable `SMFEXTENSION`. They will add that extension to any file name that does not already contain one before attempting to open it. If the setup file configures **JAM** to ignore extensions, this rule does not apply.

2. *Creating new files.* Many utilities transform files of one type to another (i.e., ASCII text to binary). These utilities must name the output file differently from the input file. They do this by appending or replacing the input file's extension. If the input file has no extension, the utility's extension is automatically appended. If the input file has an extension, the extension may be replaced or appended, depending upon the operating system. In DOS, extensions are replaced; in UNIX, extensions are appended.

3. *Creating data structures.* The utilities `f2struct`, `dd2struct`, and `bin2c` create data structures from screen files. They name the structures by removing the path and extension from the input file name.

1.1.4

Configuring File Extensions and Rules

There are three parameters that control how **JAM** uses file extensions. The default for each of these depends on your operating system.

- A flag telling whether **JAM** should recognize and replace extensions, or ignore them. This activity is controlled by the setup variable `F_EXTREC`.

- Another flag telling whether the extension should go at the beginning or the end of the file name. The default is the end of the file name. The controlling setup variable is F_EXTOPT.
- The character that separates the extension from the name (zero means no separator). The setup variable is F_EXTSEP.

Note: In previous releases, **JAM** used one setup variable, SMUSEEXT, to control the use of file extensions. This variable is supported for backwards compatibility.

You may alter any of these defaults by changing the setup variables (see Setups for Default File Extensions, in the Setup File Chapter in the **JAM** *Configuration Guide*).

The table below contains a list of the default extensions used by utility programs:

Utility	*Extension*
bin2c	NONE
bin2hex	NONE
dd2asc	dic
dd2struct	h (for C language users)
dd4to5	NO CHANGE
ddmerge	dic
ddsort	NONE
f2asc	NONE
f2dd	dic
f2struct	h (for C language users)
f4to5	NO CHANGE
formlib	NONE
jamcheck	prv (backup)
jammap	map
jpl2bin	bin
jxform	NONE
key2bin	bin
lstdd	lst

Utility	*Extension*
`lstform`	`lst`
`modkey`	`keys`
`msg2bin`	`bin`
`term2vid`	`vid`
`txt2form`	NONE
`var2bin`	`bin`
`vid2bin`	`bin`

1.1.5

Ordering of Options and Other Arguments

Most utilities take as arguments an output file, a list of input files, and some options. Options may be supplied separately (each with its own hyphen), or together (all following a single hyphen); the two commands

```
lstform -fti myscreen
lstform -f -t -i myscreen
```

are equivalent. Option letters may be either upper– or lower–case. On certain systems such as VMS and MS–DOS, where the prevalent "switch character" is / rather than – , both are supported.

Options may be placed anywhere after the utility name—that is, before, between, or after file name arguments. For consistency, however, our synopses and examples, list the options immediately after the utility name.

The ordering of arguments for input and output file names depends on the particular utility. Some use the format:

output–file input–file...

but those that perform unidirectional conversions between binary and ASCII formats are a notable exception. They always use the format:

ascii–file binary–file...

and use the command options to determine which file is the output file, and which are the input. The help banners and this manual specify the ordering of file arguments for each utility.

Chapter 2
Utility Reference Manual

This chapter contains a detailed explanation of each utility. The utilities are arranged alphabetically. A listing of the utilites grouped by type, with brief explanations, appears starting on page 1 of the introduction to this guide. An alphabetical index to the reference pages appears in a table on page 10.

2.1
CONVENTIONS USED

Each section contains the following information:

- The name and description of the utility.
- A synopsis of its usage, that is, what you type on the command line to run it.

`literal` This font is used for words you will type verbatim. In particular, we use this font for all examples. In addition, when we name a utility, its option letters, or any **JAM** library function, we use this font to distinguish it from the rest of the text.

bold We use bold italics to show where screen, file, and other variable names should appear. You should replace these with the appropriate names.

`[-`***x***`]` Like the help banners, we use square brackets to indicate the optional command flags, and to indicate where a file name is optional. If you use an optional flag, or specify an optional file name, do not type the brackets.

x...

Ellipsis indicates that the element may be repeated one or more times.

- A complete description of the utility's inputs, outputs, and processing.
- The error messages associated with the utility, and suggestions for correcting the errors.

2.2 REFERENCE

This section contains a detailed explanation of each utility. The table below serves as an index key to the reference section.

bin2c

convert binary **JAM** files to ASCII C

SYNOPSIS

`bin2c [-flv] ascii-file binary-file...`

OPTIONS

`-f` Overwrite an existing output file.

`-l` Convert file names sent to output to lower case.

`-v` Generate list of files processed.

DESCRIPTION

This program converts binary files created with other **JAM** utilities into character arrays in ASCII C. ***ascii-file*** is usually a new file name. (To overwrite an existing file, you must use the `-f` option.)

When the utility creates the ASCII C file, it will generate an array for each of the binary input files. An array in the file has the form

`char binary-file[] = { contents of file };`

where ***binary-file*** is the name of the binary file, with its path and extension stripped off. If you use the `-l` option, ***binary-file*** will be in lower case.

Files created with `bin2c` arrays may be compiled, linked with your application, and added to the memory-resident form list. (See the **JAM** *Programmer's Guide* for more information on memory-resident lists.) The following files may be made memory-resident:

- key translation files (`key2bin`)
- setup variable files (`var2bin`)
- video configuration files (`vid2bin`)
- message files (`msg2bin`)
- JPL files (`jpl2bin`)
- screen files (`jxform`)

There is no utility to convert ***ascii-file*** back 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`.

Note: On VAX/VMS, invoke this utility with the command `B2C`.

ERRORS

`Insufficient memory available.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: Try to increase the amount of available memory.

`File "%s" already exists; use '-f' to overwrite.`
Cause: You have specified an output file that already exists.
Corrective action: Use the –f flag to overwrite the file, or use another name.

`"%s": Permission denied.`
Cause: An input file was not readable, or an output file was not writeable.
Corrective action: Check the permissions of the file in question.

bin2hex

convert binary files to and from hex ASCII, for transport

SYNOPSIS

```
bin2hex -c [-flv] ascii-file binary-file...
bin2hex -x [-flv] ascii-file
```

OPTIONS

`-c`	Create an ASCII file from one or more binary file.
`-x`	Extract all binary files 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.

DESCRIPTION

The `bin2hex` utility translates binary files of any description to and from a hexadecimal ASCII representation. It is useful for transmitting files between computers.

Either the `-c` or the `-x` switch is required; the others are optional.

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***. Extensions are not stripped off. If you use `-l`, the binary file names in ***ascii-file*** will be in lower case.

With `-x`, ***ascii-file*** is the input file. The utility will extract each ***binary-file*** in ***ascii-file***, and put each file in the current directory. If you use `-l`, the binary file names will be in lower case. Selective extraction of binary files from ***ascii-file*** is not supported. Only one argument is supported with the `-x` option; any additional arguments are ignored.

Note: On VAX/VMS, invoke this utility with the command `B2HEX`.

ERRORS

`%s already exists`
`%s already exists, it is skipped`
Cause: The command you have issued would overwrite an existing output file.
Corrective action: If you are sure you want to destroy the old file, reissue the command with the `-f` option.

dd2asc

convert a data dictionary to ASCII and binary formats

SYNOPSIS

```
dd2asc -a [-f] ascii-file [binary-file]
dd2asc -b [-f] ascii-file [binary-file]
```

OPTIONS

`-a`	Creates an ASCII listing of a data dictionary file.
`-b`	Creates a data dictionary file from an ASCII listing.
`-f`	Overwrites an existing output file.

DESCRIPTION

Data dictionary files created with `jxform` are binary files. With the `-a` option, you can convert a data dictionary file to a readable listing of the data dictionary. You may modify this listing, according to certain rules, and use the utility with the `-b` option to convert the listing back to a binary file which **JAM** can use. With the `-b` option, you may also create a new data dictionary.

Either `-a` or `-b` must be used. If ***binary-file*** is not named, the utility defaults to `data.dic` (or the data dictionary initialized with the library function `sm_dicname` or with the setup file variable `SMDICNAME`).

CREATING AN ASCII FILE FROM A DATA DICTIONARY

Running `dd2asc` with the `-a` option creates a complete, readable listing of the contents of your data dictionary.

Here is an example. Assume that the data dictionary that you have created with `jxform` is called `data.dic` and contains the following:

- Built–in default values designating a simple (non–array) field with a length of 10, no character edits, a scope of 2, and display attributes of underlined, highlighted and white.
- `fld1`, a shifting and scrolling vertical array with the following attributes: length = 8, elements = 3, distance = 2, maxshifting length = 20,

increment = 1, scrolling items = 10, page size = 3, and both the circular and isolate options. A help screen is attached to it, named `help3.jam`. It has a status line text and is a shifting and scrolling array. In addition, it has a comment attached to its entry in the data dictionary.

- `fld2`, with a length of 15, a scope of 3, and display attributes of reverse video, underlined, highlighted and blue. The character edit is numeric. The field edits are right–justified and data–required. It has a currency format CURRENCY (CURRENCY is one of the default mnemonics in the message file. If it has not been changed in the message file, it corresponds to the message entry `SM_0DEF_CURR = ".22,1$"`.) This field also has a data type of double, with a precision of 2.
- `fld3`, with a length of 10, a scope of 5, and display attributes of highlighted and white. It has a time format of `HR:MIN2 AMPM`, based on a 12–hour clock, and gets the time from the operating system. It is also protected from everything except clearing. (`HR`, `MIN2`, and `AMPM` are default mnemonics from the message file).
- `fld4`, with a length of 2 and a regular expression, `[A-Z][0-9]`, as a character edit.
- `fld5`, with a length of 3 and a character edit of digits only. You have specified acceptable data entry ranges of 100–500 and 800–999.
- `fld6`, a vertical array with these attributes: length = 10 and elements = 3.
- A comment in the data dictionary, which is not attached to either a field or a record.
- A group, named `grp1`, with a checklist and box edit. The box attributes are : white, underline, and offset=2. The occurrences of `fld6` are members of the group.
- A record, with the name `outst` and a data dictionary comment, and consisting of 3 fields: `name`, with a data type of char string; `telnum`, with a data type of omit from struct; and `amount`, with a data type of float and a precision of 2.

To create an ASCII listing in a new file called `asc`, use this command:

```
dd2asc -a asc data.dic
```

Here is a listing of the output file `asc`:

```
D:
   SCOPE=2 UNFILTERED LENGTH=10 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:fld1   this is a comment
   SCOPE=2 UNFILTERED LENGTH=8 ARRAY-SIZE=3 VERT-DISTANCE=2
   MAX-LENGTH=20 SHIFT-INCR=1 MAX-OCCUR=10 PAGE-SIZE=3 CIRCULAR
ISOLATE
   WHITE UNDERLINE HILIGHT
   TEXT='hey this is status'
   HELP-SCRN=help3.jam

F:fld2
   SCOPE=3 UNFILTERED LENGTH=15 ARRAY-SIZE=1
   WHITE HILIGHT

F:fld3
   SCOPE=5 UNFILTERED LENGTH=10 ARRAY-SIZE=1
   WHITE HILIGHT
   24-HOUR SYST-DATETIME=HR:MIN2 AMPM

F:fld4
   SCOPE=2 CHAR-MASK LENGTH=2 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
   REG-EXP (CHAR)=[A-Z][0-9]

F:fld5
   SCOPE=2 DIGITS-ONLY LENGTH=3 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
   RANGE1(FROM)=100
   RANGE1(TO)=500
   RANGE2(FROM)=800
   RANGE2(TO)=999

F:fld6
   SCOPE=2 UNFILTERED LENGTH=10 ARRAY-SIZE=3
   WHITE UNDERLINE HILIGHT

# This comment is not attached to any entry in the dictionary.

G:grp1
   SCOPE=2 CHECKLIST
   MEMBERS=3
   BOX(WHITE UNDERLINE HILIGHT) OFFSET=2

R:outst
   FIELDS= name, telnum(OMIT), amount(FLOAT:2)
```

You can also create or modify a text file and use the `-b` option to turn the file into a data dictionary. To do that, you will need the rules explained in the next section.

CREATING A DATA DICTIONARY FROM AN ASCII FILE

There are five types of entries in the ASCII file, corresponding to the kinds of information in a data dictionary. They are

1. default field attributes
2. fields
3. groups
4. records
5. stand–alone comments.

The first line of each entry in the ASCII file identifies the entry as one of these types. Leading blank spaces are not permitted on an entry's first line. (Therefore, `D:`, `F:`, `G:`, `R:`, or # always begins a line at column 1). All other blank space is ignored. You may use blank lines to separate entries. Lines may be continued by using the backslash '\'.

ENTRY TYPES

Each type of entry for the data dictionary is discussed in the sections below.

Default Field

A default field entry, begins with:

```
D:
```

No other text is permitted on this line. Below this line, list the edits of the default field using the keywords for field attributes. (The next section lists these attributes.)

If an ASCII file contains an entry for a default field, this entry must be first.

The following field and group entries may be listed in any order. The comments in these entries are optional.

Fields

A field entry (other than the default) begins with

`F:` ***fieldname*** `[`***comment***`]`

fieldname is the name of the field and ***comment*** is any optional comment text you want attached to the data dictionary entry. This line is followed by a list of the field's attrib-

utes. There are keywords for all the field attributes and we describe them in a later section.

Groups

Each group entry begins with

```
G:groupname [comment]
```

groupname is the name of the group and ***comment*** is any optional comment text you want attached to the data dictionary entry. The group attributes follow. The keywords for group attributes are explained in a later section.

Records

Every record entry begins with

```
R:recordname [comment]
    FIELDS= fieldname [(data-type)],...
```

recordname is the name of the record and ***comment*** is optional. `FIELDS=` is required on the next line. It should be followed by the names of the fields in ***recordname***. Use a comma to separate the field names. You may specify a data type for ***fieldname*** by enclosing a data type keyword in parentheses and placing it after the field name.

Stand–Alone Comments

Comments which are not attached to other entries begin with

```
# comment
```

ATTRIBUTE KEYWORDS

There are two types of keywords for attributes: **flags** and **values**. A flag keyword stands by itself and needs no other information, like the `HILIGHT` display attribute. It may appear on the same line as other primary keywords. A value keyword must be accompanied by more information; it is followed by an equal sign (=), then more keywords or strings. Value keywords must appear alone on a line, accompanied only by the information attached to them.

`dd2asc` usually reads only the first few characters of each keyword, so you can truncate keywords if you wish. However, the utility itself always generates the full names, and we recommend that you do so as well, for better documentation.

The following is a list of all keywords, presented in the order in which you would encounter the attributes in the Screen Editor. For explanations of each field attribute, refer to the *Author's Guide*.

Note: There are often two or three keywords for the same attribute. For example, the field edit for right-justified may be indicated by any of the following keywords:

```
RIGHT-JUSTIFIED
RT-JUST
RTJUST
```

In this text, we list the default keywords and syntax used by **JAM**. For a complete listing of all the keywords, see the UT_2A entries in the message file.

Scope

This is a value keyword, but it may appear on the same line as other keywords. It must be followed by a number between 0 and 9. Entries with a scope of 0 are constants and cannot be changed at runtime.

```
SCOPE= Integer
```

Field Keywords

The following keywords specify all the field attributes available in the Screen Editor.

Display Attributes

All of these are flag keywords.

```
BLACK
BLUE
GREEN
CYAN
RED
MAGENTA
YELLOW
WHITE

NON-DISPLAY
REVERSE
BLINKING
UNDERLINE
HILIGHT
DIM
STANDOUT
ALTERNATE
```

Character Edits

All of these are flag keywords except for REG-EXP (CHAR).

```
UNFILTERED, ALL
DIGITS-ONLY
YES-NO
LETTERS-ONLY
NUMERIC
ALPHANUMERIC

CHAR-MASK
REG-EXP (CHAR) = expression
```

Note: Choose CHAR-MASK if you have a regular expression. The value keyword REG-EXP(CHAR) = ***expression*** must follow, on a separate line.

Field Edits

Flag and value keywords:

```
RIGHT-JUSTIFIED
REQUIRED

RETURN-ENTRY
RETCODE= integer

PROTECTED

PROTECTED FROM DATA-ENTRY \
TABBING-INTO CLEARING VALIDATION

MENU-FIELD
SUBMENU= menu-screen-name
RETCODE= integer

CLR-INPUT
UPPER-CASE
LOWER-CASE
MUST-FILL
NO-AUTOTAB
REG-EXP(FIELD)=regular-expression

NULLFLD=SM_YESstring
NULLFLD=SM_NOstring
```

If you choose RETURN-ENTRY or MENU-FIELD, you may include the value keyword RETCODE = ***integer-value*** on a separate line. Menu fields may also have a SUBMENU value keyword.

If the field is protected from everything, use PROTECTED alone. If it is only partially protected, use PROTECTED FROM followed by any or all of the four values listed.

With NULLFLD, use the value of SM_YES (e.g., y) to fill an empty field with ***string***. For example, if SM_YES is "y" in the message file, then

```
NULLFLD=y?
```

will fill the empty field with question marks. If SM_NO is "n", then

```
NULLFLD=nunknown
```

will put the string "unknown" in the empty field. The string will not be repeated.

Field Attachments

All of these are value keywords.

NEXTFLD(NORMAL) = ***next-field-designation***
NEXTFLD(ALTERNATE) = ***alternate-next-field-designation***

PREVFLD(NORMAL) = ***previous-field-designation***
PREVFLD(ALTERNATE) = ***alternate-previous-field***

HELPSCR= ***help-screen-name***
AUTO-HELP= ***automatic-help-screen-name***

ITEM_SELECT= ***item-selection-screen-name***
AUTO-ITEM= ***automatic-item-selection-screen-name***

TBL-LOOKUP= ***screen-name***

TEXT= ***field-status-string***

MEMO1= ***string***
MEMO2= ***string***
...
MEMO9= ***string***

Miscellaneous Edits

All of these are value keywords.

Field Functions:

ENTRY-FUNC= ***function-name***
VAL-FUNC= ***function-name***
EXIT-FUNC= ***function-name***

If any of these functions are JPL procedures, precede ***function-name*** with `jpl` (i.e., `jpl` ***function-name***).

Date/Time:

`12-HOUR SYST-DATETIME=` ***date-time-format***
`24-HOUR SYST-DATETIME=` ***date-time-format***
`12-HOUR USER-DATETIME=` ***date-time-format***
`24-HOUR USER-DATETIME=` ***date-time-format***

date-time-format is specified using the date/time mnemonics from the message file.

`MATH=` ***expression***
`MATH=` ***expression***`;` ***expression***`;` ***expression***`; ...  CKDIGIT=` ***modulus***
`MIN-DIGITS=` ***count***

`RANGE1(FROM) =` ***value***
`RANGE1(TO) =` ***value***
`...`
`RANGE9(FROM) =` ***value***
`RANGE9(TO) =` ***value***

`JPL-TEXT=`***jpl-program-line***

Currency:

Any or all of the following flag or value keywords may follow `CURR-FORMAT=`

`LOCAL-FORMAT-NO=` ***integer*** `(between 1 and 10)`
`DEC-SYMBOL=` ***c***
`MIN-DEC-PLACES=` ***integer***
`MAX-DEC-PLACES=` ***integer***
`THOU-SEP-SYMBOL=` ***c***
`CURR-SYMBOL=` ***ccccc***
`CUR-LEFT`
`CUR-RIGHT`
`CURR-MIDDLE`
`ROUND-UP`
`ROUND-DOWN`
`ROUND-ADJUST`
`FILL-CHAR=` ***c***
`RIGHT-JUST`
`LEFT-JUST`
`CLEAR-IF-ZERO`
`APPLY-IF-EMPTY`

c represents a single character. `CURR-SYMBOL` will accept up to 5 characters.

Note: Ten currency mnemonics are defined in message entries FM_0MN_CURRDEF to FM_9MN_CURRDEF, and their ten corresponding token formats are defined in message entries SM_0DEF_CURR to SM_9DEF_CURR. dd2asc does not recognize the message file mnemonics or formats as flag or value keywords. Instead, a message file currency format is specified by the value keyword LOCAL_FORMAT_NO= and an integer between one and ten. For compatibility with previous releases, the 2asc utilities number the currency formats between 1 and 10, rather than 0 and 9 like the message tags. Therefore, LOCAL-FORMAT-NO= 1 refers to mnemonic assigned to FM_0MN_CURRDEF (CURRENCY by default) and the token format assigned to SM_0DEF_CURR (".22,1$" by default). LOCAL-FORMAT-NO= 10 refers to the mnemonic assigned to FM_9MN_CURRDEF (DEFAULT9 by default) and the token format assigned to SM_9DEF_CURR (".09" by default). If you assign LOCAL-FORMAT-NO an integer outside the range of 1 to 10, it will be ignored and no currency format will be made to the field.

Size

Most of these are value keywords, but all may appear on the same line as other keywords.

LENGTH= ***onscreen-length***
ARRAY-SIZE= ***number-of-onscreen-elements***
VERT-DISTANCE= ***offset***
HORIZ-DISTANCE= ***offset***
WORD-WRAP
ALT-SCROLL-FUNC= ***function-name***
MAX-LENGTH= ***shifting-length***
SHIFT-INCR= ***count***
MAX-ITEM= ***number-of-occurrences***
PAGE-SIZE= ***number***
CIRCULAR
ISOLATE

Data Type

One value keyword, FTYPE=, which may take on any one of the following values:

OMIT
CHAR-STR
INT
UNSIGNED
SIGNED
SHORT
LONG

FLOAT: ***integer***
DOUBLE: ***integer***
ZONED: ***integer***
PACKED: ***integer***

If you choose FLOAT, DOUBLE, ZONED, or PACKED, you may follow it with an optional colon and integer, designating the precision. For example, the following specifies a floating point number with three decimal places:

```
FTYPE=FLOAT:3
```

In addition, you may indicate SIGNED or UNSIGNED with the types ZONED and PACKED. For example, the following specifies a zoned number with two decimal places and unsigned edit:

```
FTYPE=ZONED:2, UNSIGNED
```

Group Keywords

RADIO-BUTTON
CHECKLIST
MEMBERS=***integer***
BOX (***display attributes***)
OFFSET=***integer***
BOUNCE-BAR
AUTO-TAB
FTYPE=***data type***

Display attributes for a box are enclosed in parentheses. See the list of display attributes under Field Keywords. A group may also have a data type edit. See the list of data types under Field Keywords.

ERRORS

ASCII file syntax errors do not stop the creation of a data dictionary. The errors and anything following them on the same line are skipped. However, all valid entries preceding them on the same line, and all entries on lines without errors, are incorporated into the data dictionary.

`Can't read %s.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the file in question.

`Can't open %s.`
Cause: An output file could not be created, due to lack of permission or perhaps disk space.
Corrective action: Correct the file system problem and retry the operation.

`%s is not a valid data dictionary.`
Cause: The binary file you have named in the data dictionary parameter does appear to be not have the correct format (as determined by a special code placed at the beginning of the file).
Corrective action: Check the file with the Data Dictionary Editor.

`Error writing %s.`
Cause: The utility incurred an I/O error while processing the file named in the message.
Corrective action: Retry the operation.

`%s already exists.`
Cause: You have specified an existing output file.
Corrective action: Use the –f option to overwrite the file, or use a different name.

`Bad data in %s.`
Cause: A binary input file is corrupt.
Corrective action: Make sure the file is of the correct type.

There are also numerous messages regarding syntax errors in an ASCII input file, which are intended to be self–explanatory.

dd2struct

convert data dictionary records to programming language data structures

SYNOPSIS

```
dd2struct [-flpv] [-o file] [-g lang] dictionary [record-name ...]
dd2struct [-flpv] [-e ext]  [-g lang] dictionary [record-name...]
```

OPTIONS

`-f`	Overwrites an existing output file.
`-l`	Converts structure names to lower case.
`-p`	Creates the output files in the same directory as the data dictionary.
`-v`	Lists all structures written.
`-o`	Places all the structures in a single output file, whose name is supplied with the option. `-o-` (without ***outfile***) prints the output to the display.
`-e`	Creates individual structures this extension. The default extension is "h". If `-o` is entered, `-e` will be ignored.
`-g`	Creates the structures in the programming language whose name follows the option letter. `C` is the default. `CB` creates `C` structures with blank-filled string.

DESCRIPTION

This utility reads in a data dictionary, and creates programming language data structures corresponding to some or all of the records defined in that data dictionary. You may use the output of this utility to declare a structure and to define variables in your application program. In addition, several **JAM** library functions read from and write to data structures. If you are using `sm_rrecord` or `sm_wrecord`, you can use this utility to create the required data structure from data dictionary records.

By default, this utility creates C language structures with null-terminated strings. If you need blank-filled rather than null-terminated strings, use the `-g` option with language `CB`. The utility will then create C data structures with blank-filled strings.

A data dictionary name must be specified. **JAM** will not supply a default for ***dictionary.***

If there are no ***record-name*** arguments, all records in the dictionary will be used; otherwise, only the selected records will be used. If the data dictionary contains no re-

cords, **JAM** displays an error message saying there are no records to convert and `dd2struct` generates no output. If a record contains a field which is not in the data dictionary, **JAM** displays an error message omits the field from the structure.

If output is generated, the files will each contain one structure corresponding to a record in the data dictionary. A file is named after a record and has the extension `h` or the extension specified by the option `-e`. The `-o` option may be used to save all the structures in a single output file.

The format in an output file is the following:

```
struct record-name {
        type    field-name;
        type    field-name;
        ...
        type    field-name;
};
```

The structure tag is the name of the record. Unless the type `omit` is specified, or the field or group is not in the data dictionary, a field or group in a record ***record-name*** is a member of the structure ***record-name***. The data type of a structure member is derived according to the following rules:

1. If any type edit except for `char` is specified for a field or group in the Record Window of the Data Dictionary Editor, this data type is used in the structure. The record field type may be different that the data dictionary field type. This allows a field to be processed as different types. A non-`char` type overrides a data dictionary type when a structure is generated.
2. If no type was specified in the Record Window, but the data dictionary element has a data type edit, this type is used. Please note that some character edits automatically change the data type edit. In particular, a field with a "digits-only" character edit has the default type `unsigned int`. A field with a "numeric" character edit, has the default type `double`.
3. A group has the default type `unsigned int`.
4. All other fields are of type `char`.

If a field has multiple occurrences, the corresponding structure member will be declared as an array.

By default, only "C" data types are supported in `jxform`. They are the following:

`jxform` *data type*	*C data type*
`omit`	
`char string`	`char`

jxform *data type*	*C data type*
int	int
unsigned int	unsigned int
short int	short
long int	long
float	float
double	double
zoned dec.	char
packed dec.	char

ERRORS

`Language %s undefined.`
Cause: The language you have given with the `-g` option has not been defined in the utility's tables.
Corrective action: Check the spelling of the option.

`%s already exists.`
Cause: You have specified an existing output file.
Corrective action: Use the `-f` option to overwrite the file, or use a different name.

`%s has an invalid file format.`
Cause: An input file is not of the expected type.
Corrective action: Check the spelling and type of the offending file.

`'%s' has no data to convert.`
Cause: An input file is empty, or does not have the names you specified.
Corrective action: Check the names.

`Not enough memory to process '%s'.`
`Unable to allocate memory.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: In non–virtual memory environments, try to increase the amount of available physical memory.

dd3to5

convert Release 3 data dictionaries to Release 5 format

SYNOPSIS

```
dd3to5 [-fpx] [-e ext] dictionary...
dd3to5 [-fpx] [-o file] dictionary...
```

OPTIONS

`-f`	Overwrites an existing file of the same name. Usually not recommended.
`-p`	Creates the output file in the same directory as the input file. Usually not recommended.
`-x`	Deletes extension from Release 3 data dictionaries.
`-e`	Appends ***ext*** to the output file.
`-o`	Writes output to the named file. Only one dictionary argument is permitted with this option. `-o` overrides `-e` and `-x`.

DESCRIPTION

This utility is provided to developers upgrading from **JAM** Release 3.

`dd3to5` reads in Release 3 data dictionaries, converts them to Release 5 format, and writes them out according to the option given.

We strongly discourage overwriting an original Release 3 data dictionary. Rather, run the utility from another directory and specify the full pathname to the dictionary. You might also run the utility with the `-o` option. For example,

```
dd3to5 -o data5.dic data.dic
```

would create `data5.dic` as a release 5 data dictionary without overwriting the original.

ERRORS

Please see `dd4to5`.

dd4to5

convert Release 4 data dictionaries to Release 5 format

SYNOPSIS

```
dd4to5 [-fp] [-e ext] dictionary...
```

OPTIONS

`-f`	Overwrite an existing file of the same name. Usually not recommended.
`-p`	Create the output file in the same directory as the input file. Usually not recommended.
`-e`	Appends ***ext*** to the output file name.

DESCRIPTION

This utility reads in **JAM** Release 4 data dictionaries, converts them to Release 5 format, and writes them out with the same names.

We strongly discourage overwriting an original Release 4 data dictionary. Rather, run the utility from another directory and specify the full pathname to the dictionary. You might also run the utility with the `-e` option. For example,

```
dd4to5 -e new data.dic
```

would create `data.dic.new` as a release 5 data dictionary without overwriting the original.

ERRORS

`At least one data dictionary name is required.`
Cause: Utility does not default to `SMDICNAME`.
Corrective action: Specify a dictionary name.

`File` ***dictionary*** `already exists. Use '-f' to overwrite.`
Cause: The specified dictionary file (including the extension you are appending to release 5 files) already exists in the directory from which you are running `dd4to5`.
Corrective action: Run the utility from another directory, or run the utility specifying a different extension, or run the utility using the `-f` option.

dictionary `is not a Release 4 data dictionary.`
Cause: The specified dictionary file is not a Release 4 dictionary.
Corrective action: If the file is a Release 3 dictionary, use the `dd3to5` utility. dd3to5 uses the same options as dd4to5.

dictionary `is a release 5 file.`
Cause: The specified ***dictionary*** was already converted to release 5, or was originally created with release 5.
Corrective action: Utility does not need to be run.

ddmerge

combine binary data dictionaries

SYNOPSIS

`ddmerge [-f]` ***destination dictionary...***

OPTIONS

`-f` Overwrites existing file ***destination.***

DESCRIPTION

This utility combines two or more binary data dictionaries into one. Use it, to build a data dictionary from simpler components in a modular fashion.

The utility reads the first ***dictionary*** into memory. As it processes each subsequent ***dictionary,*** `ddmerge` adds new entries to the file in memory. A new entry has a unique field, group, or record name. If an entry is an exact duplicate of one already added, `ddmerge` ignores the new entry. (Comment text is not compared.) If an entry has the same name but different edits or contents, `ddmerge` displays a message describing the difference; it does not save the variant entry. After processing the last entry, `ddmerge` writes out the new binary data dictionary to ***destination.*** You may use the utility `jamcheck` to update screens with the entries in ***destination.***

Since the merging is done in memory, the maximum size allowed for ***destination*** is machine–dependent.

You may change the name of the default data dictionary via the environment variable `SMDICNAME`. See the *Configuration Guide* for details.

ERRORS

See `f2dd`.

ddsort

sort data dictionary entries by name

SYNOPSIS

```
ddsort [-f] output-file dictionary
```

OPTIONS

`-f` Overwrites existing dictionary file.

DESCRIPTION

This utility alphabetically sorts the elements of a data dictionary. Elements include records, fields, and groups. Fields within records are not sorted.

ERRORS

One destination and one source file name required.
Cause: Either ***output-file*** or ***dictionary*** was not specified.
Corrective action: Name both arguments.

Format Error in ***dictionary.***
Cause: Either source file is not a data dictionary or the file is corrupted.
Corrective action: Check that ***dictionary*** is a valid data dictionary.

f2asc

convert screens between binary and editable ASCII format

SYNOPSIS

```
f2asc -a [-cf] ascii-file screen...
f2asc -b [-f] ascii-file
```

OPTIONS

-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.

DESCRIPTION

jxform creates binary screen files. You may use f2asc with -a to create an ASCII listing of a screen's contents and edits (much like dd2asc), modify the file, and convert it back to a binary screen file using f2asc with -b.

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**.

The utility is provided for the following purposes.

- You can document in English the contents of a screen (field and group names, edits, etc. . .)
- You can create an ASCII listing of a screen and use a text editor for global searching and replacing. When your edits are complete you can convert the listing to a binary screen file.
- You can use source code control systems (i.e., SCCS or RCS in UNIX) to manage your screens.

The text files generated by f2asc are a complete description of the contents of the screen. The text is self– explanatory. However if you wish to edit the file or create one from scratch, you will need to understand the mnemonics used in the file and the rules for organizing them.

EDITING OR CREATING A SCREEN

There are five types of entries in the ASCII file. They are

- screen characteristics
- default field attributes
- fields
- groups
- comments.

ENTRY TYPES

The first line of each entry in the ASCII file identifies the entry as one of these types. Leading blank spaces are not permitted on an entry's first line. (Therefore, `S:`, `D:`, `F:`, `G:`, or # always begins a line at column 1). All other blank space is ignored. You may use blank lines to separate entries.

Screen Characteristics

A screen entry begins with

`S:` ***screenname***

screenname is the name of the screen. This is the first entry for every screen in ***asciifile.***

Default Field Symbols

A default field entry begins with

`D:SYMBOL=` ***field-symbol***

Below this line, is a list of the edits of the default field using the keywords for field attributes. (The next section lists these attributes.)

If an ASCII file contains entries for default fields, these entries precede the entries for fields, groups, and records. A screen may contain up to nine different default field symbols.

Fields

A field entry (other than the default) begins with

F : ***fieldname***

fieldname is the name of the field. This line is followed by a list of the field's attributes. There are keywords for field attributes, which are described in a later section.

Groups

Each group entry begins with

G : ***groupname***

groupname is the name of the group. The group attributes follow. The keywords for group attributes are explained in a later section.

Stand–Alone Comments

Comment lines begin with

\# ***comment***

When `f2asc -a` is used, **JAM** gives each field's number in a comment line. If the field is an array, it lists the field number of each element in the array. You can use the `-c` option turn this feature off.

ATTRIBUTE KEYWORDS

There are two types of keywords for field attributes: **flags** and **values**. A flag keyword stands by itself and needs no other information, like the `HILIGHT` display attribute. It may appear on the same line as other primary keywords. A value keyword must be accompanied by more information; it is followed by an equal sign (=), then more keywords or strings. Value keywords must appear alone on a line, accompanied only by the information attached to them.

`f2asc` usually reads only the first few characters of each keyword, so you can truncate keywords if you wish. However, the utility itself always generates the full names, and we recommend that you do so as well, for better documentation.

The following is a list of all keywords, presented in the order in which you would encounter the attributes in `jxform`. For explanations of each attribute, refer to the *Author's Guide*.

Note: There are often two or three keywords for the same attribute. For example, the field edit for right–justified may be indicated by any of the following keywords:

```
RIGHT-JUSTIFIED
RT-JUST
RTJUST
```

In this text we list the default keywords and syntax used by **JAM**. For a complete listing of all the keywords, see the UT_2A entries in the message file.

SCREEN ATTRIBUTES

These are flag and value keywords:

LINES=***integer***
COLUMNS=***integer***
BACKGROUND=(***attributes***)
BORDER=(***attributes***)
STYLE***integer***
DEFAULT-ATT=(***attributes***)
KEYSET= ***filename***
DISPLAY(***row, column***) (***attributes***) (***length***) =***text or graphics***
MENU-MODE

attributes are enclosed in parentheses. See the list of attribute keywords below. For display text, you must indicate in parentheses integers for the beginning row and column of the text.

FIELD ATTRIBUTES

The following keywords specify all the field attributes in jxform.

Display Attributes

All of these are flag keywords.

BLACK
BLUE
GREEN
CYAN
RED
MAGENTA
YELLOW
WHITE

NON-DISPLAY
REVERSE
BLINKING
UNDERLINE
HILIGHT

```
DIM
STANDOUT
ALTERNATE
```

Character Edits

All of these are flag keywords.

```
UNFILTERED
DIGITS-ONLY
YES-NO
LETTERS-ONLY, ALPHABETIC
NUMERIC
ALPHANUMERIC

CHAR-MASK
REG-EXP(CHAR) = expression
```

Note: Choose CHAR-MASK if you have a regular expression. The value keyword REG-EXP(CHAR) = ***expression*** must follow, on a separate line.

Field Edits

Flag and value keywords:

```
RIGHT-JUSTIFIED
REQUIRED
RETURN-ENTRY
RETCODE= integer

PROTECTED

PROTECTED FROM DATA-ENTRY \
TABBING-INTO CLEARING VALIDATION

MENU-FIELD
SUBMENU= menu-screen-name
RETCODE= integer

CLR-INPUT
UPPER-CASE
LOWER-CASE
MUST-FILL
NO-AUTOTAB
REG-EXP(FIELD) =regular-expression
```

NULLFLD=***SM_YES string***
NULLFLD=***SM_NO string***

If you choose RETURN-ENTRY or MENU-FIELD, you may include the value keyword RETCODE = ***integer*** on a separate line. Menu fields may also have a SUBMENU value keyword.

If the field is protected from everything, use PROTECTED alone. If it is only partially protected, use PROTECTED FROM followed by any or all of the four values listed.

With NULLFLD, use the value of SM_YES (e.g., y) to fill an empty field with ***string***. For example,

```
NULLFLD=y *
```

to fill an empty field with asterisks, or

```
NULLFLD=n unknown
```

to put the string "unknown" in an empty field.

Field Attachments

All of these are value keywords.

NEXTFLD(NORMAL) = ***next-field-designation***
NEXTFLD(ALTERNATE) = ***alternate-next-field-designation***

PREVFLD(NORMAL) = ***previous-field-designation***
PREVFLD(ALTERNATE) = ***alternate-previous-field***

HELP-SCRN= ***help-screen-name***
AUTO-HELP= ***automatic-help-screen-name***
ITEM_SELECT= ***item-selection-screen-name***
AUTO-ITEM= ***automatic-item-selection-screen-name***

TBL-LOOKUP= ***screen-name***

TEXT=***field-status-string***

MEMO1= ***string***
MEMO2= ***string***
. . .
MEMO9= ***string***

Miscellaneous Edits

All of these are value keywords.

Field Functions:

ENTRY-FUNC= ***function-name***
VAL-FUNC= ***function-name***
EXIT-FUNC= ***function-name***

If any of these functions are JPL procedures, precede ***function-name*** with `jpl` (i.e., `jpl` ***function-name***).

Date/Time:

12-HOUR SYST-DATETIME= ***date-time-format***
24-HOUR SYST-DATETIME= ***date-time-format***
12-HOUR USER-DATETIME= ***date-time-format***
24-HOUR USER-DATETIME= ***date-time-format***

MATH= ***expression***
MATH= ***expression; expression; expression; ...***

CKDIGIT= ***sum***
MIN-DIGITS= ***count***

RANGE 1(FROM) = ***value***
RANGE 1(TO) = ***value***
...
RANGE 9(FROM) = ***value***
RANGE 9(TO) = ***value***
JPL-TEXT=***jpl-program-line***

Currency format:

CURR-FORMAT=

Any or all of the following flag or value keywords may follow CURR-FORMAT=

LOCAL-FORMAT-NO= ***integer*** (between 1 and 10)
DEC-SYMBOL= ***c***
MIN-DEC-PLACES= ***integer***
MAX-DEC-PLACES= ***integer***
THOU-SEP-SYMBOL= ***c***
CURR-SYMBOL= ***ccccc***
CUR-LEFT
CUR-RIGHT
CURR-MIDDLE
ROUND-UP
ROUND-DOWN
ROUND-ADJUST

FILL-CHAR= ***c***
RIGHT-JUST
LEFT-JUST
CLEAR-IF-ZERO
APPLY-IF-EMPTY

c indicates a single character. CURR-SYMBOL will accept up to 5 characters.

Note: Ten currency mnemonics are defined in message entries FM_0MN_CURRDEF to FM_9MN_CURRDEF, and their ten corresponding token formats are defined in message entries SM_0DEF_CURR to SM_9DEF_CURR. f2asc does not recognize the message file mnemonics or formats as flag or value keywords. Instead, a message file currency format is specified by the value keyword LOCAL_FORMAT_NO= and an integer between one and ten. For compatibility with previous releases, the 2asc utilities number the currency formats between 1 and 10, rather than 0 and 9 like the message tags. Therefore, LOCAL-FORMAT-NO= 1 refers to mnemonic assigned to FM_0MN_CURRDEF (CURRENCY by default) and the token format assigned to SM_0DEF_CURR (".22,1$" by default). LOCAL-FORMAT-NO= 10 refers to the mnemonic assigned to FM_9MN_CURRDEF (DEFAULT9 by default) and the token format assigned to SM_9DEF_CURR (".09" by default). If you assign LOCAL-FORMAT-NO an integer outside the range of 1 to 10, it will be ignored and no currency format will be made to the field.

Size

Most of these are value keywords, but all may appear on the same line as other keywords.

LENGTH= ***onscreen-length***
ARRAY-SIZE= ***number-of-onscreen-elements***
VERT-DISTANCE= ***offset***
HORIZ-DISTANCE= ***offset***
WORD-WRAP
ALT-SCROLL-FUNC= ***function-name***
MAX-LENGTH= ***shifting-length***
SHIFT-INCR= ***count***
MAX-ITEM= ***number-of-occurrences***
PAGE-SIZE= ***number***
CIRCULAR
ISOLATE

Position

All of these are value keywords. They describe a fields position on the screen.

LINE= ***integer***
COLUMN= ***integer***

Data Type

One value keyword, FTYPE, which may take on any one of the following values:

OMIT
CHAR-STR
INT
UNSIGNED
SIGNED
SHORT
LONG
FLOAT:***precision***
DOUBLE:***precision***
ZONED:***precision***
PACKED:***precision***

If you choose FLOAT, DOUBLE, ZONED, or PACKED, you may follow it with an optional colon and number, designating the precision. For example, the following specifies a floating point with three decimal places:

```
FTYPE=FLOAT:3
```

In addition, you may indicate SIGNED or UNSIGNED with types ZONED and PACKED. For example, the following specifies a zoned number with two decimal places and an unsigned edit.

```
FTYPE=ZONED:2, UNSIGNED
```

Groups

RADIO-BUTTON
CHECKLIST
BOX(***attribute***)
OFFSET= ***integer***
BOUNCE-BAR
AUTO-TAB
FTYPE= ***type***
OCCUR ***number***= string
SELECTED-OCCUR=***integer***

Display attributes for a box edit are enclosed in parentheses.

ERRORS

ASCII file syntax errors do not stop the creation of a screen file. The errors and anything following them on the same line are skipped. However, all valid entries preceding

them on the same line, and all entries on lines without errors, are incorporated into the screen file.

`Can't read %s.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the file in question.

`Can't open %s.`
Cause: An output file could not be created, due to lack of permission or perhaps disk space.
Corrective action: Correct the file system problem and retry the operation.

`Error writing %s.`
Cause: The utility incurred an I/O error while processing the file named in the message.
Corrective action: Retry the operation.

`%s already exists.`
Cause: You have specified an existing output file.
Corrective action: Use the –f option to overwrite the file, or use a different name.

`Bad data in %s.`
Cause: A binary input file is corrupt.
Corrective action: Make sure the file is of the correct type.

There are also numerous messages regarding syntax errors in an ASCII input file, which are intended to be self–explanatory.

f2dd

create or update a data dictionary from screen files

SYNOPSIS

```
f2dd [-v] dictionary screen...
```

OPTIONS

`-v` Generate list of screens processed.

`-l` Convert screen names sent to data dictionary to lowercase. This utility creates a data dictionary record for each screen. Use this option to force all screen record names to lowercase. This option does not affect the case of the individual field or group names within the record.

DESCRIPTION

If a new data dictionary is being created, the utility:

- Creates a record for each named screen and enters the record in the data dictionary. Each record contains the names of all fields and groups on the screen except for screen name fields;
- Enters every field which appears on the specified screen into the data dictionary, along with the field's characteristics.
- Enters every group which appears on the specified screen into the data dictionary, along with each group's attributes.

If a data dictionary is being updated, the utility:

- Creates records for any of the specified screens, if the records are not already in the data dictionary;
- Adds entries for fields and groups not already in the data dictionary;
- Lists all differences between the information in the data dictionary and the specified screens;
- Lists all fields and groups in the specified screens which have the same name but different field characteristics.

ERRORS

`Unable to allocate memory.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

```
Too many entries for data dictionary.
Too many data dictionary entries.
Too many entries for LDB.
```
Cause: The output file has reached the maximum possible size.
Corrective action: Specify fewer inputs, or remove unnecessary fields from them.

```
Can't read form %s.
Bad data in form %s.
%s is not a form.
```
Cause: An input file was missing, unreadable, or not the right kind.
Corrective action: Check the spelling, presence, and permissions of the file in question.

```
Form %s has no fields.
Form %s has no named fields.
```
Cause: Warning only. The screen will make no contribution to the output.
Corrective action: None.

```
Can't create record "%s" -- same name as data dictionary
Field.
Can't add field "%s" in %s -- same name as data dictionary
Record.
```
Cause: A screen or field has a name that conflicts with something already in the data dictionary.
Corrective action: Rename one of the items.

```
Record "%s" in %s differs from data dictionary record.
Field "%s" in %s differs from data dictionary field.
Field "%s" in %s has different edits from data dictionary
field.
```
Cause: Warning only. A screen or screen field differs from a similarly named item already in the data dictionary. The latter will be retained.
Corrective action: Rename one of the items.

```
Can't write %s.
Can't write destination file.
```
Cause: An output file could not be created, due to lack of permission or perhaps disk space.
Corrective action: Correct the file system problem and retry the operation.

f2struct

create program data structures from screens

SYNOPSIS

```
f2struct [-flpv] [-o outfile] [-g lang] screen...
f2struct [-flpv] [-e ext]     [-g lang] screen...
```

OPTIONS

`-f`	Overwrites an existing output file.
`-l`	Converts structure names to lower case.
`-p`	Creates each output file in the same directory as the corresponding input file.
`-v`	Lists all structures written.
`-o`	Causes all output to be placed in ***outfile***. –o– (without ***outfile*** sends output to the screen instead of a file.
`-e`	Appends the ***extension*** to all output files. If `-o` is also used, `-e` is ignored. "h" is the default extension when "C" language is used.
`-g`	Creates the structures in the named programming language. The default is "C". Use `CB` for blank– filled strings.

DESCRIPTION

This program creates program source files containing data structure definitions matching the input files. If you use the `-o` option, ***outfile*** will contain a structure for each named ***screen***. If `-o` is not used, `f2struct` will create a file for each screen; each file will contain one structure.

By default, this utility creates C language structures with null–terminated strings. If you need blank–filled rather than null–terminated strings, use the `-g` option with language `CB`. This utility will then create C data structures with blank–filled strings.

The format in the output file is the following:

```
struct screen–name {
        type field–name;
        type group–name;
        ...
        type field–name;
};
```

The structure tag is the name of the screen file with its extension stripped off. Unless a field has the data type `omit`, every field on the screen is a member of the structure. Please note that array fields created with the group option from the "Create Special Objects" menu have a default type of `omit`. If a field has no name, then `fld`*m* is used, where *m* is the field number. Every group on the screen is member of the structure. Checklists are declared as arrays.

The data types for the structure members are derived according to the following rules:

1. If a field has a data type edit, its type is used.
2. If a field has no data type edit but has a "digits–only" character edit, its type is `unsigned int`. A field with no data type, but a "numeric" character edit, has type `double`.
3. A group has the default type `unsigned int`. Another type may be assigned by using the "Group Attributes" window.
4. All other fields are of type `char`.

If a field has multiple occurrences, the corresponding structure member will be declared as an array.

`jxform` supports C data types. In addition, it provides zoned decimal and packed decimal for developers using the **JAM** Cobol Language Interface. See the table below.

`jxform` *data type*	*C data type*
`omit`	
`char string`	`char`
`int`	`int`
`unsigned int`	`unsigned int`
`short int`	`short`
`long int`	`long`
`float`	`float`
`double`	`double`
`zoned dec.`	`char`
`packed dec.`	`char`

If `omit` is chosen as a data type, the field or group will be ignored by `f2struct`.

ERRORS

`Language %s undefined.`
Cause: The language you have given with the `-g` option has not been defined in the utility's tables.
Corrective action: Check the spelling of the option.

`%s already exists.`
Cause: You have specified an existing output file.
Corrective action: Use the `-f` option to overwrite the file, or use a different name.

`%s has an invalid file format.`
Cause: An input file is not of the expected type.
Corrective action: Check the spelling and type of the offending file.

`'%s' has no data to convert.`
Cause: An input file is empty, or does not have the names you specified.
Corrective action: Check the names.

`Not enough memory to process '%s'.`
`Unable to allocate memory.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

`At least one screen name is required.`
Cause: You have not given any screen files as input.
Corrective action: Supply one or more screen file names.

f3to5

convert Release 3 screens to Release 5 format

SYNOPSIS

```
f3to5 [-adjlu1] [-fpvx] [-e ext] screen...
```

OPTIONS

`-a`	Do not convert `jam_pf1` field to `AUTO` **JAM** control string.
`-d`	Do not remove `jam_d_dflt` and `jam_f_dflt` fields.
`-j`	Do not convert **JAM** control fields to control strings.
`-l`	Interpret Release 3 HIGH/LOW intensity as LOW (default is HIGH).
`-u`	Convert all unprotected fields to menu fields (for item selection).
`-1`	Do not convert `jam_first` to screen entry function.
`-f`	Overwrite existing files with the same name. Usually not recommended.
`-p`	Create each output file in the same directory as the corresponding input file. This option is not recommended.
`-v`	Print the name of each screen as it is processed.
`-x`	Delete extension form Release 3 files.
`-e`	Appends ***ext*** to the names of all output files.

DESCRIPTION

This utility is provided to developers upgrading from **JAM** Release 3.

`f3to5` converts one or more Release 3 screens to Release 5 format. It gives each new screen the same name as the old one, unless you use the option `-e`. If you do not wish to use this option, we recommended that you run this utility in a directory other than the one storing the original Release 3 screens.

ERRORS

Please see `f4to5`.

f4to5

convert Release 4 screens to Release 5 format

SYNOPSIS

```
f4to5 [-fpv] [-e ext] screen...
```

OPTIONS

`-f`	Overwrites existing files with the same name. Usually not recommended.
`-p`	Creates each output file in the same directory as the corresponding input file. This option is not recommended.
`-v`	Prints the name of each screen as it is processed.
`-e`	Appends ***ext*** to the names of all output files.

DESCRIPTION

`f4to5` converts one or more Release 4 screens to Release 5 format. It gives each new screen the same name as the old one, unless you use the option `-e`. If you do not wish to use this option, we recommended that you run this utility in a directory other than the one storing the original Release 4 screens.

`f4to5` will convert date, time, and currency fields to Release 5 formats. Release 4 screens with these formats must be converted for these fields to work properly. All other screens can be included in Release 5 applications without conversion by this utility. You can, of course, run the utility on all Release 4 screens if it is more convenient than isolating those that contain date, time, or currency fields.

ERRORS

`At least one screen name is required.`
Cause: You failed to provide the name of a screen to be converted.
Corrective action: Re–type the command, followed by the name of at least one Release 4 screen.

`File` ***screen*** `already exists. Use '-f' to overwrite.`
Cause: The specified screen file (including the extension you are appending to release 5 files) already exists in the directory from which you are running `f4to5`.
Corrective action: Run the utility from another directory, or run the utility specifying a different extension, or run the utility using the `-f` option.

screen `is not a release 4 screen.`
*Cause:*The specified screen file is not a release 4 screen.
Corrective action: If the file is a release 3 screen, use the `f3to5` utility.

screen `is a release 5 screen.`
Cause: The specified ***screen*** was already converted to release 5, or was originally created with release 5.
Corrective action: None needed.

formlib

application librarian

SYNOPSIS

```
formlib -c [-flv] library [file...]
formlib -d [-lv] library file...
formlib -r [-lv] library file...
formlib -t [-l] library [file...]
formlib -x [-flv] library [path/file...]
```

OPTIONS

`-c` Creates a new library and puts all named files in the library. If no files are named, all files in the current directory will be put in the library.

`-d` Deletes the named files from the library.

`-r` Adds or replaces the named files in ***library***.

`-t` Lists the current contents of ***library***.

`-x` Extracts one or more files from ***library*** and places them in the specified directory, or in the current directory if no directory is specified. If no files are named, everything in ***library*** will be extracted.

`-f` Overwrites an existing library or file with the same name.

`-l` Uses lower case.

`-v` Prints the name of each file as it is being processed.

DESCRIPTION

`formlib` creates libraries where you may store screens and binary JPL files. (ASCII JPL files are converted to binary with the utility `jpl2bin`.) You may actually store any type of file in a library, but **JAM** will retrieve only screen and binary JPL files from a library at runtime. With `formlib` you can store many screens in a single file and not clutter a directory with individual screen and JPL files.

Exactly one of the options `-c`, `-d`, `-r`, `-t`, or `-x`, and a library name must be specified to execute `formlib`.

When specifying ***file***, you may use any of the wildcard or pattern matching symbols supported by your operating system. For example on MS–DOS, the command

```
formlib -c screenlib *.jam
```

will put all files with the extension `jam` in the library `screenlib`.

The option `-l` is useful for developers whose operating systems distinguish between upper and lower case file names. If `-c` or `-r` is used with `-l` all the specified files added to the library will have lowercase names. For example on a UNIX system,

```
formlib -cl newlib *
```

puts every file which is in the current directory into the library `newlib`. All the files in the library are named in lowercase. Therefore if `MAIN.JAM` is a file in the current directory, the library file is named `main.jam`.

When `-l` is used with `-d`, `-t`, or `-x`, **JAM** will convert file names to lower case when looking for them in the library. For example,

```
formlib -xl lib SCRN2
```

tries to extract `scrn2` from `lib`. If it finds the screen, it will create the output file `SCRN2`.

ERRORS

`Library '%s' already exists; use '-f' to overwrite.`
Cause: You have specified an existing output file.
Corrective action: Use the –f option to overwrite the file, or use a different name.

`Cannot open '%s'.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the file in question.

`Unable to allocate memory.`
`Insufficient memory available.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

`File '%s' is not a library.`
Cause: The named file is not a form library (incorrect magic number).
Corrective action: Check the spelling and existence of your library.

`'%s' not in library.`
`No forms in library.`
Cause: A screen you have named is not in the library.
Corrective action: List the library to see what's in it, then retry the operation.

`Temporary file '%s' not removed.`
Cause: The intermediate output file was not removed, probably because of an error renaming it to the real output file.
Corrective action: Check the permissions and condition of the files, then retry the operation.

jamcheck

update screens to match entries in a data dictionary

SYNOPSIS

```
jamcheck [-cix] [-fp] [-e ext] [-v] dictionary screen...
jamcheck [-m] [-cix] [-fp] [-e ext] [-v] dictionary screen...
jamcheck [-abdglqrstz] [-cix] [-fp] [-e ext] [-v] dictionary screen...
```

OPTIONS

For matching:

`-m` Matches on everything.

`-a` Matches on field display attributes.

`-b` Matches on all group attributes.

`-d` Matches on field character edits.*

`-g` Matches on field miscellaneous edits.

`-l` Matches on field formatting edits, such as date, time, and currency format.*

`-q` Matches on field protection.

`-r` Matches on field functions (entry, validation, and exit).

`-s` Matches on field length and number of occurrences.*

`-t` Matches on field status text.

`-z` Matches on field help and item selection edits.

* Defaults if no matching options are specified.

For Changing Fields and Groups to Match Data Dictionary:

`-c` Changes field characteristics and group attributes to the values in the data dictionary, according to the specified or default matching options. Old screens will be saved with an extension `prv` or ***ext*** specified with `-e`.

`-i` Requests confirmation before making changing to a screen field. Changes made according to the specified or the default matching options.

`-x` Extends onscreen length and/or array size of field. By default, a screen field is made larger by making it shifting or scrolling.

For Screen Backups:

`-e` Appends ***ext*** rather than `prv` as the extension to backup screen files.

`-f` Overwrites existing screen backup files.

`-p` Places the backup screens in the same directory as the originals, rather than the current directory.

Other:

`-v` Lists screen names as they are processed.

DESCRIPTION

This utility reads a data dictionary into memory, compares it with one or more screens. It compares screen fields against data dictionary entries with the same names. Command options control which of the many field or group characteristics are checked. If no options are given, the utility checks field character edits, field format commands, and field size, as though the options were `-dls`.

This utility can also change the screen fields to bring them into conformity with the data dictionary. It will change field characteristics according to the specified matching options, or the default options if none were specified. **JAM** saves the old screens with the extension `prv`, or ext if `-e` is used.

If you tell `jamcheck` to expand fields onscreen with `-x` and the screen cannot accommodate a larger field, the field will be made shifting or scrolling; fields will always be extended, offscreen if necessary. Fields can always be made smaller.

Screens without named fields are listed, but otherwise ignored. Screen and field names without corresponding data dictionary entries are also ignored.

ERRORS

`Unable to allocate memory.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

`Can't read %s.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the file in question.

`%s is not a valid data dictionary.`
`Bad data in %s.`
Cause: An input file was of the wrong kind, or has been corrupted.
Corrective action: Check the type of the indicated file.

`File %s already exists; use '-f' to overwrite.`
Cause: You have specified an existing output file.
Corrective action: Use the –f option to overwrite the file, or use a different name.

`Field "%s" in %s has same name as data dictionary Record.`
Cause: Warning only. The indicated field will not be compared.
Corrective action: None.

There are also many informational messages, which are meant to be self–explanatory.

jammap

list relations among **JAM** screens

SYNOPSIS

```
jammap [-clmrsw] [-pv] [-e ext] topscreen
jammap [-clmrsw] [-pv] [-o file] topscreen
```

OPTIONS

`-c` The Control String Function Report provides an alphabetic listing of functions which are called from **JAM** control strings attached to logical keys (in the Control String Window, not in menu fields). Control string functions begin with a caret.

`-l` The Linkage Report shows the contents of every **JAM** control field for each screen in ***topscreen***'s directory. The screens are listed in alphabetical order. This top–level screen as well as forms and windows referenced by display–form or display–window control strings are included in this report. Control strings that reference a screen not in the list will be flagged.

`-m` The Links Missing Report lists the names of screens that are referenced by control links, but are not found in the current directory.

`-r` This report lists all the screens checked.

`-s` The System Call Report lists programs and commands included in **JAM** control strings beginning with an exclamation point.

`-w` The List of Parameter Windows contains names of all the parameter windows included in **JAM** control strings via the percent sign ("%") option.

`-p` Places backup files in the same directory as the original screen files.

`-v` Lists input screens and processing steps to the terminal as they occur.

`-e` Gives the map file the extension that follows the option letter. If `-o` is used, `-e` will be ignored.

`-o` Places the output listing in the file whose name follows the option letter. –o– (without ***mapfile***) displays the listing on the terminal.

DESCRIPTION

`jammap` reports on the status of the screens in a **JAM** directory and the relationships among them. You must give specify the name of the top–level screen. It scans the direc-

tory containing ***topscreen*** and creates reports according to the specified options. By default, the reports are placed in a file with the name of the top–level screen and an extension of .map, or ***ext*** if `-e` is used.

`jammap` produces a report with 6 sections. You can select which sections you want printed by specifying the one or more of the options above. If you specify none of these options, `jammap` will print all 6 sections.

ERRORS

`Exactly 1 form name is required.`
Cause: The argument to this utility is the top–level screen of a **JAM** application; you have supplied extra parameters.
Corrective action: Retry the command, without the excess.

`Unable to allocate memory.`
`Insufficient memory for lists, form`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

`Can't find top level form`
Cause: The input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the file in question.

jpl2bin

compile a JPL text file into a binary file

SYNOPSIS

```
jpl2bin [-pv] [-e ext] JPL-file...
```

OPTIONS

`-p`	Places output files in the same directory as input files.
`-v`	Generates a list of files processed.
`-e`	Appends ***ext*** as extension to output file names (default is `bin`).

DESCRIPTION

JPL modules created with the Screen Editor in a JPL procedure window are compiled when the module is saved (by pressing XMIT in the JPL procedure window). JPL modules stored in ASCII text files are compiled at runtime, each time the module is called. You can use `jpl2bin` to eliminate runtime compilation of JPL files. This utility compiles a JPL text file and saves it to a binary file. The binary file may be called directly.

If you want to add JPL files to libraries or memory resident lists, you must first convert the file with `jpl2bin`. JPL routines in binary files may be placed in libraries by themselves or in libraries with related screen files.

The compilation process performs syntax checking on command words, converts JPL command words into tokens, partitions the file into procedures. Please refer to the *JPL Guide* for a more detailed discussion of the JPL compilation process.

ERROR CONDITIONS

`%s: No such file or directory.`
Cause: The input file could not be found.
Corrective action: Check the spelling and presence of the file or the full pathname of the input file.

`Unrecognized verb.`
Cause: Line in the JPL file does not begin with a valid JPL command.
Corrective action: Check the spelling of the command word in question.

`USAGE: FOR varname = Value WHILE ( expression ) STEP [+-] value`
Cause: A `for` statement in the JPL file is incorrect.

Corrective action: Check that the statement contains the three command words `for`, `while`, and `step`.

`Verb needs arguments.`
Cause: JPL command requires at least one argument.
Corrective action: See reference section in *JPL Guide* for information on the command in question.

key2bin

convert key translation files to binary format

SYNOPSIS

```
key2bin [-pv] [-e ext] keyfile...
```

OPTIONS

`-p`	Places the binary files in the same directories as the input files.
`-v`	Lists the name of each input file as it is processed.
`-e`	Appends ***ext*** to the output file name. The default extension is `bin`.

DESCRIPTION

The `key2bin` utility converts key translation files into a binary format for use by applications using the **JAM** library. The key translation files themselves may be generated by JYACC modkey, which is documented elsewhere in this chapter, or created with a text editor according to the rules described in the chapter on key files in the **JAM** *Configuration Guide*.

keyfile is the name of an ASCII key translation file. By convention it is an abbreviation of the terminal's name, plus a tag identifying it as a key translation file; for instance, the key translation file for the vt100 is called `vt100keys`. The utility first tries to open its input file with the exact name you put on the command line; if that fails, it appends `keys` to the name and tries again. The output file will be given the name of the successfully opened input file, with a default extension of `bin`.

To make a key translation file memory–resident, first run the binary file produced by this utility through the `bin2c` utility to produce a program source file; then compile that file and link it with your program.

ERRORS

`File '%s' not found`
`Neither '%s' nor '%s' found.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the file in question.

`Unknown mnemonic in line: '%s'`
Cause: The line printed in the message does not begin with a logical key mnemonic.
Corrective action: Refer to smkeys.h for a list of mnemonics, and correct the input.

`No key definitions in file '%s'`
Cause: Warning only. The input file was empty or contained only comments.
Corrective action: None.

`Malloc error`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

`Cannot create '%s'`
`Error writing '%s'`
Cause: An output file could not be created, due to lack of permission or perhaps disk space.
Corrective action: Correct the file system problem and retry the operation.

lstdd

list the contents of a data dictionary

SYNOPSIS

```
lstdd [-cdglr] [-p] [-e ext]  [dictionary]
lstdd [-cdglr] [-p] [-o file] [dictionary]
```

OPTIONS

`-c`	Lists comments.
`-d`	Lists default field characteristics for new entries.
`-g`	Lists attributes for all groups.
`-l`	Lists field characteristics for all entries.
`-r`	Lists the fields belonging to data dictionary records.
`-p`	Places the listing in the same directory as the input file.
`-e`	Append ***ext*** to the output file name. The default is `lst`. If `-o` is used, `-e` is ignored.
`-o`	Places the output in the file whose name follows the option letter. The default is the name of the data dictionary with the extension `lst`. –o– (without ***file***) displays the output to the terminal.

DESCRIPTION

This utility reads a data dictionary, by default `data.dic`, and creates a human–readable listing of the contents. By default, all information in the dictionary is listed, but you may use one or more options to make a selective listing.

An output file created with `lstdd` cannot be converted to a binary data dictionary. Use `dd2asc` for this purpose.

ERRORS

`Error opening input file.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the file in question.

`Error opening output file.`
Cause: An output file could not be created, due to lack of permission or perhaps disk

space.
Corrective action: Correct the file system problem and retry the operation.

```
Unable to allocate memory.
Can't allocate memory.
```
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

```
Error reading data dictionary file.
Error writing list file.
```
Cause: The utility incurred an I/O error while processing the file named in the message.
Corrective action: Retry the operation.

```
Invalid file format or incorrect version.
%s is not a valid data dictionary.
Bad data in %s.
```
Cause: An input file has the wrong magic number or is corrupt.
Corrective action: Make sure all the input files are data dictionaries. If you have Release 3 data dictionaries, you may need to run dd2r4 to update them.

lstform

list selected portions of screens

SYNOPSIS

```
lstform [-abdgijmnpstvx] [-k num ] [-e ext] screen...
lstform [-abdgijmnpstvx] [-k num ] [-o file] screen...

lstform [-abdgijmnpstvx] [-w] [-e ext] screen...
lstform [-abdgijmnpstvx] [-w] [-o file] screen...
```

OPTIONS

`-a` Lists default field characteristics for the screen.

`-b` Excludes line numbers.

`-d` Lists display data.

`-e` Outputs a file for each named screen.

`-g` Lists all group edits.

`-i` Lists initial field data, including offscreen data.

`-j` Lists **JAM** control strings.

`-k` Permits lines of an arbitrary length, ***num***. If ***num*** = 0, columns are wrapped and not truncated. If ***num*** is 20 or greater, the utility truncates columns following column ***num***. For values of ***num*** greater than 0 but less than 20, the utility defaults to the value 20.

`-m` Lists data relevant to the screen as a whole: border, screen entry function, etc.

`-n` Includes a snapshot of the screen showing underscores in place of fields.

`-p` Places output files in the same directory as the corresponding inputs.

`-s` Includes a snapshot of screen showing display data and initial onscreen contents of fields.

`-t` Lists all field edits.

`-v` Prints the name of each screen on the terminal as it is processed.

`-x` Excludes form feeds and page numbers.

`-k` Does not truncate columns. Permits lines of an arbitrary length, ***num***.

`-w` Wraps columns over 76 or 77, rather than truncating.

`-e` Generates one output file, appending ***ext*** to the file name. The default extension is `lst`. If the option `-o` is used, `-e` is ignored.

`-o` Sends the output to a single file whose name follows the option letter. `-o-` (without ***file***) sends output to screen.

DESCRIPTION

This program lists all or some edits of one or more screen files. By default, all the data about each field in each screen is included. Using command options, however, you can direct that only some of the display be generated.

By default, screen snapshots are truncated at 80 columns. Unless the `-b` option is used, the default snapshot shows 76 columns of the screen, and saves 4 columns for line numbers. Use `-b` to exclude line numbers, or use `-w` to wrap columns, or use `-k` to specify the number of columns to be shown.

ERRORS

`Error opening input file.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the file in question.

`Error opening output file.`
Cause: An output file could not be created, due to lack of permission or perhaps disk space.
Corrective action: Correct the file system problem and retry the operation.

`Unable to allocate memory.`
`Can't allocate memory.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

`Error reading form file.`
`Error writing list file.`
Cause: The utility incurred an I/O error while processing the file named in the message.
Corrective action: Retry the operation.

modkey

key translation file editor

SYNOPSIS

```
modkey [keyfile]
```

DESCRIPTION

The `modkey` utility provides a convenient mechanism for specifying how keys on a particular keyboard should operate in the **JAM** environment. It provides for defining the function, editing, and cursor control keys used by **JAM**, as well as soft keys and keys that produce foreign or graphics characters. Finally, modkey can store label text corresponding to your keys, for use in prompts and help messages.

The output of modkey is a text file called the key translation file. After being converted into a binary table by the `key2bin` utility, it is used to translate physical characters generated by the keyboard into logical values used by the **JAM** library. By dealing with logical keys, programs can work transparently with a multitude of keyboards.

Refer to the *Author's Guide* for a table explaining the functions of the cursor control and editing keys. The format of the key translation file generated by `modkey` is explained in the chapter on key files in the **JAM** *Configuration Guide*.

KEY TRANSLATION

The ASCII character set is comprised of eight–bit characters in the range 0 to 255 (hex FF). It defines characters in the ranges hex 20 to hex 7E and hex A0 to hex FE as data characters, and the rest as control characters. Control characters have mnemonic names; the character hex 1B, for instance, is usually called ESC or escape. See the chapter on key files in the **JAM** *Configuration Guide* for a listing. Note that certain computers, such as PRIME, "flip" the high bit of ASCII characters; on such computers, ESC would be hex 9B and the letter A would be hex C1. In this document, standard ASCII values will be used.

When you press a key, the keyboard generates either a single ASCII data character, or a sequence of characters beginning with an ASCII control code. **JAM** converts these characters into logical keys before processing them. Logical keys are numbers between zero and 65535. Logical values between 1 and hex FF represent displayable data; values between hex 100 and hex 1FF are cursor control and editing keys; values greater

than hex 1FF are function keys. Zero is never used. For a list of logical values, see the key file chapter in the JAM *Configuration Guide*.

Data characters received from the keyboard are not translated. Sequences beginning with a control character are translated to a logical value, representing a data character or function key, according to the following algorithm.

When a control character is received, we search the key translation table for a sequence beginning with that character. If there is one, we read additional characters until a match with an entire sequence in the table is found, and return the logical value from the table. If the initial character is in the table but the whole sequence is not, the whole is discarded, on the assumption that it represents a function key that is missing from the table. Finally, if a control character does not begin any sequence in the table, it is returned unchanged; this is useful for machines such as IBM PC's that use control codes for displayable characters. The *Programmer's Guide* contains a detailed discussion of key translation.

EXECUTING THE UTILITY

You execute `modkey` by typing its name on the command line, optionally followed by the name of the key translation file you want to create, examine, or change. If you are editing or examining an existing file, you should run `modkey` in the file's directory (usually the `config` directory). If you supply a key translation file name, the main menu (Figure 2) appears at once. If you do not give a file name, the welcome screen (Figure 1) appears, and you may enter one there.

CONTROL KEYS AND DATA KEYS

Since `modkey` is used to define the cursor control, editing, and function keys, these keys do not operate in the utility. Instead, displayable data keys are used for these purposes. For example, the TAB key is usually used to move the cursor from one field to the next. But since TAB is one of the keys being defined with this utility, it cannot first be recognized; the data key `t` is used instead.

Using data keys for control purposes poses no problem since, in this utility, data keys may not begin a control sequence. This will become clearer when the screens in subsequent sections are described. The control functions that are supported in the `modkey` utility and the keys that are used to provide them are given in the following table:

Control Function	*Key*
TRANSMIT	+
EXIT	-
HELP	?
REDRAW SCREEN	!
BACKSPACE	<
BACKTAB	b
FIELD ERASE	d
ENTER KEYTOP	k
TAB	t
ERASE ALL UNPROTECTED	z

The k key, or ENTER KEYTOP, causes a small window to appear under the cursor in which you may enter the label found on the key in question on your keyboard. This label will be stored in the key translation file; it can be accessed by library functions and in status line messages, and is very useful in help messages telling an operator which key to press. It operates in all the screens below the main menu that are actually used for defining keys.

WELCOME SCREEN

When you invoke modkey without supplying a file name, the welcome screen (Figure 1) is displayed as shown below.

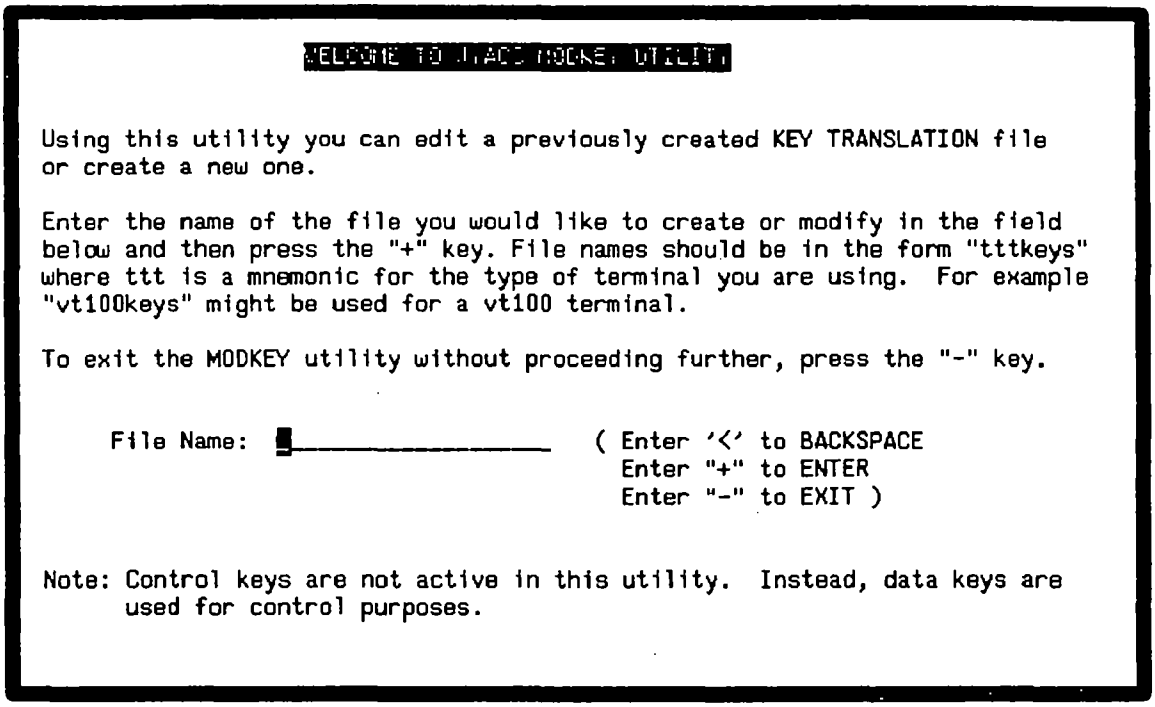


Figure 1: modkey Welcome Screen

Here you specify the key translation file to be created or modified, by entering it in the field labeled File Name. If you make a mistake, backspace over it using the < key. When finished, complete the screen by pressing the + key.

Key translation file names should begin with a mnemonic for the type of terminal you are using, and end with keys. For example vt100keys might be used for a vt100 terminal. This convention, while not mandatory, helps avoid confusion with video files and with other key translation files. All files distributed by JYACC adhere to it.

If the file already exists, it is read into memory and may be modified; otherwise, you start from scratch. All modifications are made in memory, and file updates are performed only at the conclusion of the program and at your explicit request.

To exit the modkey utility while the welcome screen is displayed, press the "–" key (EXIT).

MAIN MENU

The main menu shown in Figure 2 is displayed upon entry to the utility, and whenever you return from a lower–level screen.

```
    JYACC MODKEY UTILITY MAIN MENU

    0.  Exit
    1.  Help
    2.  Define Cursor Control And Editing Keys
    3.  Define Function Keys
    4.  Define Shifted Function Keys
    5.  Define Application Function Keys
    6.  Define Softkeys
    7.  Define Miscellaneous Keys
    8.  Test Key Translation File


Enter the desired option (0-8): _
```

Figure 2: `modkey` Main Menu Screen

You select an option by typing the corresponding number. For example, to test the key translation file, press "8". If you make an invalid selection, an error message will appear; acknowledge it by pressing the space bar.

The functions on the main menu are described in subsequent sections.

EXITING THE UTILITY

To exit modkey, press 0 on the main menu. This causes the exit screen (Figure 3) to be invoked. This screen initially contains a single field into which you enter s, e, or –.

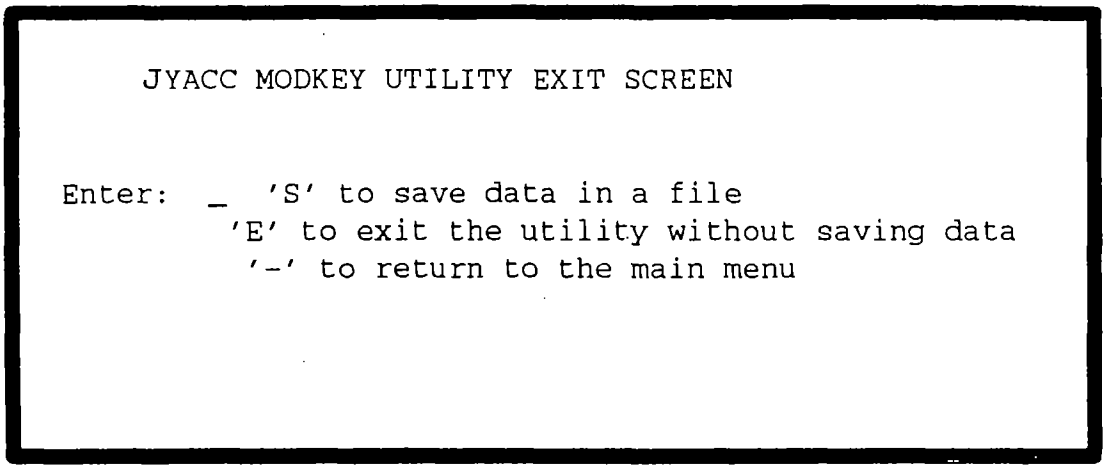


Figure 3: modkey Exit Screen

To save the key translation file on disk, enter S or s. When this is done, the file name entered in the Welcome Screen appears; you may change it if you wish, and press + to write it to disk. To exit the utility without saving the file, enter e. If you press –, the main menu will reappear, and you may make additional changes to the key translation file.

HELP SCREEN

The help screen may be selected from the main menu by pressing "1"; it appears in Figure 4.

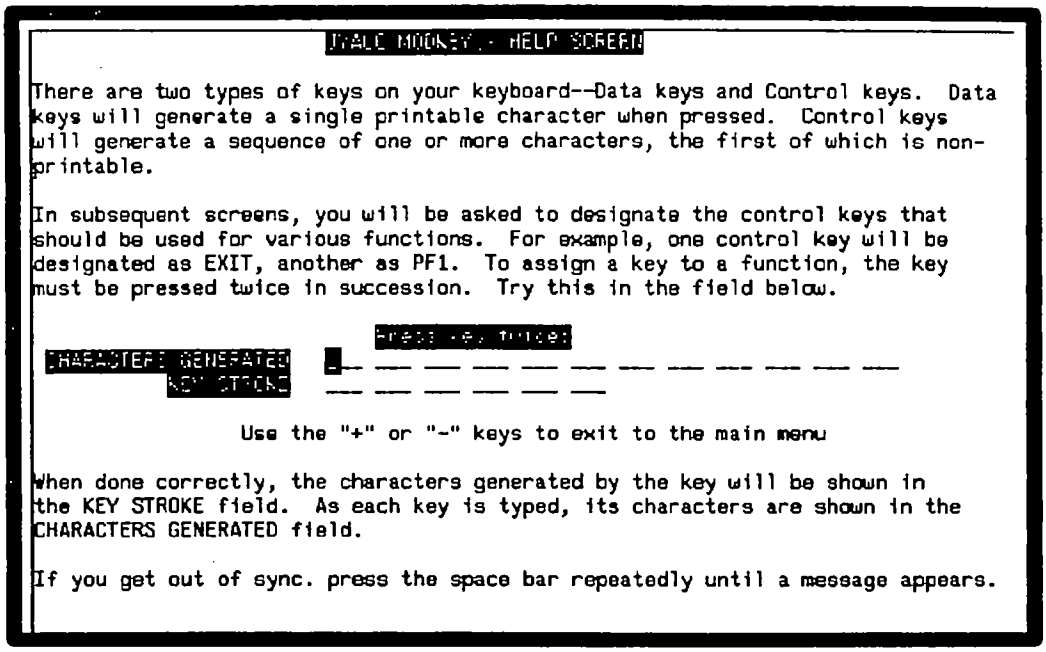


Figure 4: `modkey` Help Screen

In addition to displaying useful information, this screen may be used to test out the kinds of keystroke entry that will be required on subsequent screens in this utility.

There are two types of keys: those that generate a single ASCII character, and those that generate a sequence of characters. When a sequence is generated, the first character is always an ASCII control character. To see the characters generated by a particular key, type that key twice while the help screen is displayed. (Different keys generate different numbers of characters; when you press the key twice, the program can sense the pattern.)

When the key is pressed the first time, the characters produced will be shown following `CHARACTERS GENERATED`. When the key is pressed the second time and recognized, the sequence representing the key will appear following `KEY STROKE`.

It is sometimes desirable to designate a sequence of keystrokes to serve a particular purpose. For example, on a system with a small number of function keys, one may choose to implement the function keys F1 through F9 with the sequence `control-F` ***n*** where ***n*** is a single digit. This sequence of keystrokes can be interpreted by **JAM** as a single key.

To demonstrate this, type `control-F 1` into the help screen, by pressing the `F` key while holding the `CTRL` key down, releasing both, and then pressing the `1` key. The

sequence ACK 1 will appear following CHARACTERS GENERATED. Repeating the sequence will duplicate the ACK 1 after CHARACTERS GENERATED column, and also display ACK 1 following KEY STROKE.

If a printable ASCII character is pressed twice in a sequence, modkey immediately displays it in the KEYSTROKE column. If a non–printable character is pressed and then a second, different character is pressed, modkey will assume that a sequence is being tried and will continue displaying these characters in the CHARACTERS GENERATED field. However, if the sequence gets to be longer than six characters without starting to repeat, modkey will display Sequence too long. You must acknowledge this message by pressing the space bar. If you realize you have made a mistake in entering a key or key sequence and do not wish to duplicate it, press any key repeatedly until you see Sequence too long. After acknowledging the message, you can start over.

Note: While the help screen allows you enter a printable ASCII character as the first keypress, the other screens in modkey do not. In addition, the help screen may allow certain keystrokes to generate characters that are not permitted in other modkey screens, such as SP when the space bar is pressed.

To exit from the help screen and return to the main menu, press the "–" key (EXIT) as the first character in a sequence.

DEFINING CURSOR CONTROL AND EDITING KEYS

This function allows the operator to specify the keys that should be used for the various cursor control and editing operations. When "2" is selected from the main menu, the screen shown in Figure 5 appears. This screen has a field for each of the cursor control and editing functions supported by **JAM**. Each function has a logical value defined in the file `smkeys.h`. The purpose of this screen is to allow the operator to specify a sequence of characters for each function key.

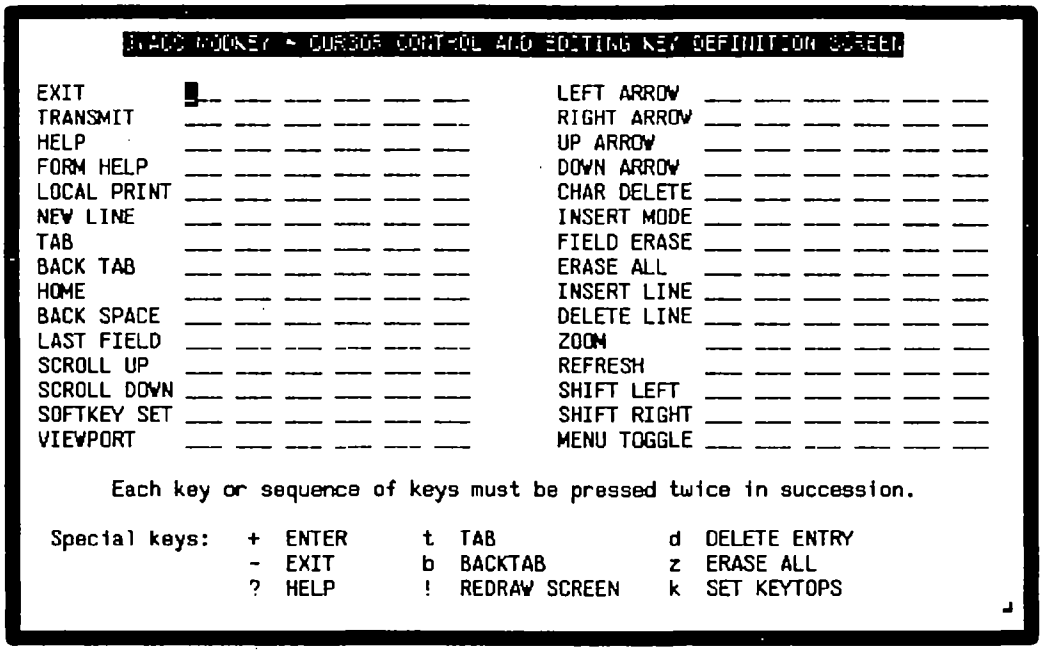


Figure 5: `modkey` Cursor Control and Editing Screen

ASSIGNING A KEY TO A FUNCTION

To designate a key for a particular cursor control or editing function, position the cursor after that function's name and press the key twice. For example, to designate a key as the EXIT key, press it twice in succession while the cursor is in the EXIT field. When `modkey` recognizes the second keystroke, the sequence of characters generated by the key will be displayed, and the cursor will move to the next field.

It is not permissible to define a printable ASCII character as a cursor control or editing key. This means that the sequence of characters generated by the key must start with an ASCII control character. If this is not the case, an error will be displayed. An error will also be displayed if the sequence of characters matches a sequence assigned to another function.

When a field is left empty, its corresponding function will not operate in programs using the keyboard translation file being defined. If your program has no use for a particular key (such as EMOH—last field on the screen), you may leave that entry blank on this screen. However, certain keys are required for the proper operation of `jxform`, and should be specified if you are creating a table for use with it. A list of the required keys is given in the key file chapter in the *Configuration Guide*.

Situations may arise in which you do not press the same key twice in succession. This will be evident because `modkey` will not display the characters that were generated. To recover, press the space bar repeatedly until the message `Sequence too long` appears. Then, after acknowledging the message with the space bar, you may enter the correct keystrokes.

To define a key label or keytop for any key on this screen, press `k` with the cursor at the beginning of the key sequence. A small, borderless window will appear, bearing the word `KEYTOP:`. In the following field, you should type whatever appears on top of the key on your keyboard, using the < key to rub out mistakes. When done, press + to save the label, or – to discard it. We recommend the use of key labels. These labels can be retrieved at run–time in user messages with `%K` and with **JAM** library routines. In addition, if someone wishes to examine your key translation file using `modkey`, they can use the keytop label rather than the codes to learn a mapping.

ASSIGNING A SEQUENCE OF KEYS TO A FUNCTION

It is sometimes desirable to designate a sequence of keystrokes to serve a particular purpose. For example, on a keyboard with few function keys, one might implement the function keys `PF1` through `PF9` with the sequences `control-F1` through `control-F9`.

One assigns a sequence of keystrokes to a function in much the same way as one assigns individual keys. The sequence is entered once in its entirety and is then repeated. Upon successful completion, the characters generated on behalf of the sequence are displayed. If you do not press the same key sequence twice, `modkey` will not display the generated characters. To recover, press the space bar repeatedly until the message `Sequence too long` appears. At this point, you may enter the correct keystrokes.

DEFINING FUNCTION KEYS

This function allows the operator to specify the keys that should be used as the function keys (`PF1 - PF24`). When "3" is selected from the main menu, the screen of Figure 6 appears.

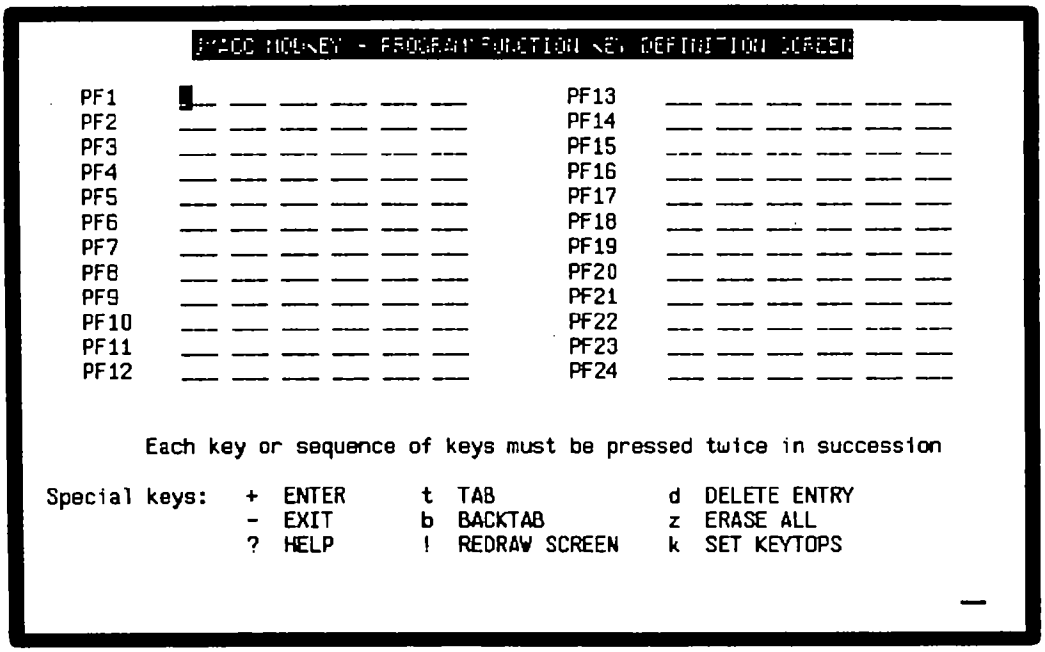


Figure 6: `modkey` Function Key Screen

This function works exactly like its counterpart for defining the cursor control and editing keys described on page 75. You designate a key or key sequence as a function key by pressing it twice, with the cursor in the field to which the sequence applies. For example, to define `control-F` as the `PF2` key, position the cursor to the `PF2` field using `t` and `b`, and type `control-F` twice in succession.

To save the changes made in this screen and return to the main menu, press the + key. To return to the main menu without saving changes, use the "–" key.

To define a key label or keytop for any key on this screen, press `k` with the cursor at the beginning of the key sequence. A small, borderless window will appear, bearing the word `KEYTOP:`. In the following field, you should type whatever appears on top of the key on your keyboard, using the < key to rub out mistakes. When done, press + to save the label, or – to discard it.

Note: If you wish to use `F11` and `F12` on a PC, you must add the `XKEY` flag to the video file.

DEFINING SHIFTED FUNCTION KEYS

This function allows the operator to specify the keys that should be used as the shifted function keys (SPF1 – SPF24). When 4 is selected from the main menu, the screen depicted in Figure 7 appears.

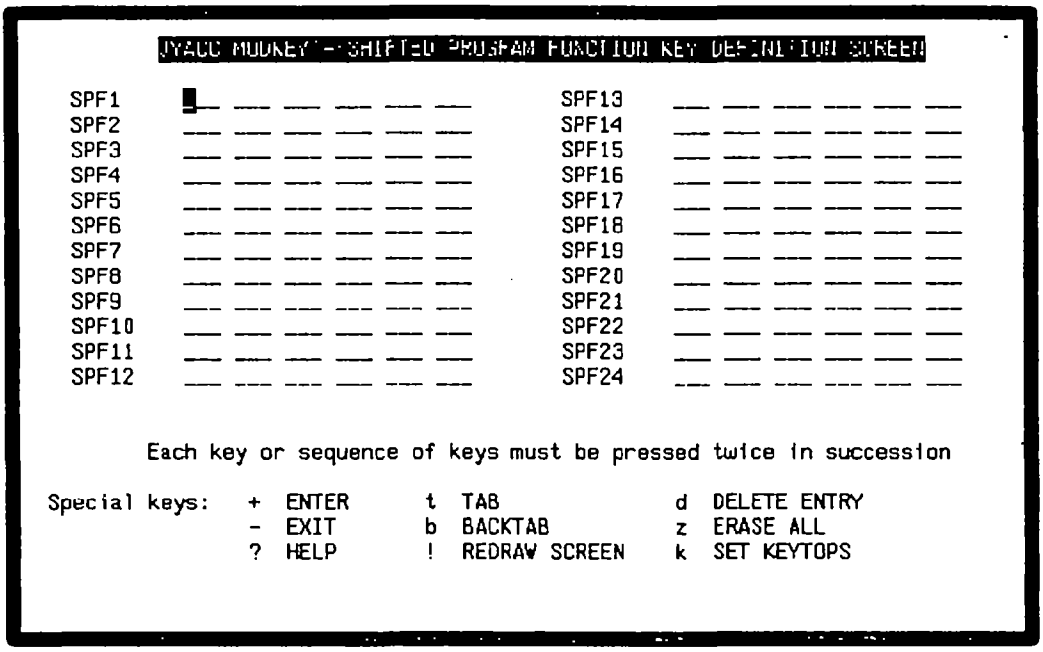


Figure 7: `modkey` Shifted Function Key Screen

This function works exactly like its counterpart for defining the keys described on page 75. You designate a key (or key sequence) as a shifted function key by pressing it twice with the cursor in the field to which the sequence applies. For example, to define the sequence of keys control–B 2 as the shifted PF2 key, position the cursor to the SPF2 field, using t and b, and type control–B 2 twice.

To save changes made in this screen and return to the main menu, press the + key as the first character in a sequence. To return to the main menu without saving the changes, use the "–" key.

To define a key label or keytop for any key on this screen, press `k` with the cursor at the beginning of the key sequence. A small, borderless window will appear, bearing the word `KEYTOP:`. In the following field, you should type whatever appears on top of the key on your keyboard, using the < key to rub out mistakes. When done, press + to save the label, or – to discard it.

DEFINING APPLICATION FUNCTION KEYS

This function allows the operator to specify the keys that should be used as the application function keys (APP1 - APP24). These logical keys are often used internally in programs without being assigned to the physical keyboard. When "5" is selected from the main menu, the screen of Figure 8 appears.

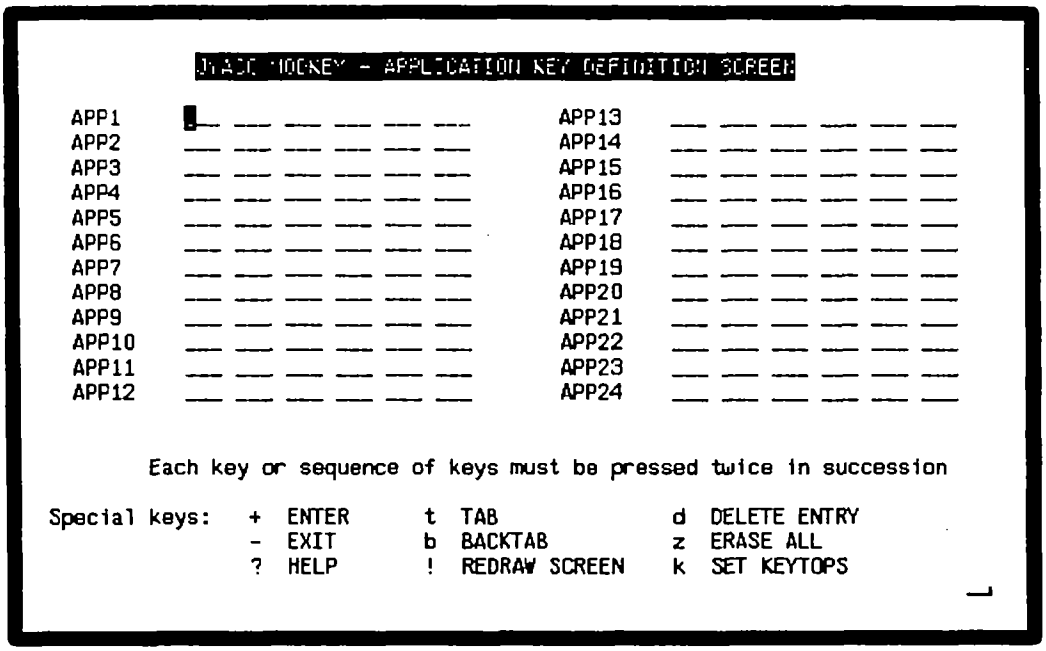


Figure 8: modkey Application Function Key Screen

This function works exactly like its counterpart for defining the keys described on page 75.

To define a key label or keytop for any key on this screen, press k with the cursor at the beginning of the key sequence. A small, borderless window will appear, bearing the word KEYTOP:. In the following field, you should type whatever appears on top of the key on your keyboard, using the < key to rub out mistakes. When done, press + to save the label, or - to discard it.

DEFINING SOFT KEYS

When "6" is selected from the main menu, the screen of Figure 9 appears.

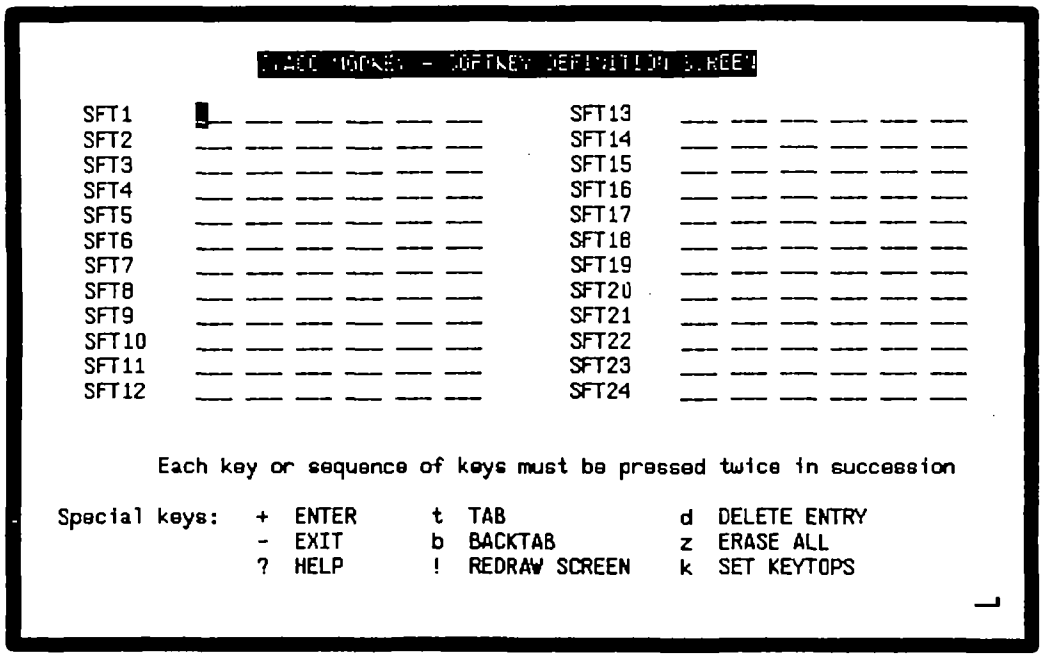


Figure 9: `modkey` Soft Key Screen

If your terminal supports soft keys, they may be defined on this screen. If you do not have hardware support for soft keys, you may define specific keystroke sequences as your soft keys.

This function works exactly like its counterpart for defining the keys described on page 75.

At run–time the logical key SFTS is used to toggle If no soft keys are defined on this screen, the defaults are the `PF` keys.

DEFINING MISCELLANEOUS KEYS

On this screen, you can specify alternate control sequences for **JAM** logical keys. When "7" is selected from the main menu, the screen of Figure 10 appears.

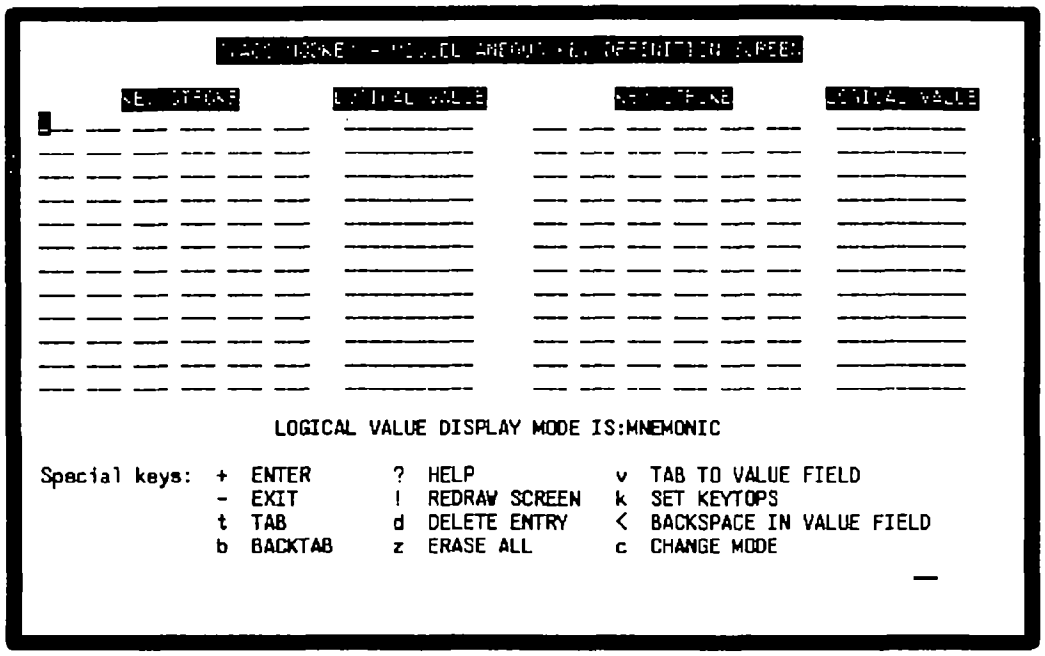


Figure 10: `modkey` Miscellaneous Key Screen

This function works in a similar manner to its counterpart for defining the cursor control and editing keys described on page 75. However, on this screen you must define the logical values as well as the sequences that produce them. (On all other screens, the logical value was implicitly determined by the field with which the sequence was associated.)

The miscellaneous key definition screen has two columns for each key being defined, labeled `KEY STROKE` and `LOGICAL VALUE`. You enter a key or key sequence into these fields twice in succession, and `modkey` displays the generated characters; then the cursor moves to the `LOGICAL VALUE` column for that key. Here, you must enter the logical value to be returned when **JAM** recognizes the sequence of characters you have just entered. You may get to the logical value field directly by pressing `v` in the corresponding `KEY STROKE` field. The existing logical value is erased from the field when you press `v`.

You may define up to 240 miscellaneous keys. Use `n` to move to the next page of miscellaneous keys, and `p` to move to the previous page.

ENTERING THE LOGICAL VALUE

Logical values are numbers, so you will be entering printable ASCII data into this field. This is unlike most other fields, where data characters are not allowed or are given special meaning (such as "b" representing BACKTAB). When entering logical values, three keys are allowed in addition to the data keys necessary to enter the value:

- The + key (TRANSMIT) signifies that the logical value just typed is correct and should be used. When it is pressed, `modkey` will first check the logical value for errors. If no errors are detected, the cursor will tab to the next `KEY STROKE` field; otherwise, an error message will appear.
- The – key (EXIT) means that the logical value just typed is incorrect and should be ignored. The cursor will go back to KEY STROKE field associated with the logical value, and the logical value is reset to its previous value. If the logical value field was previously empty, it and the `KEY STROKE` field are both cleared.
- The < key (BACKSPACE) backs up the cursor one position at a time, so that corrections to the logical value can be made. It erases previously entered data as it moves.

LOGICAL VALUE DISPLAY AND ENTRY MODES

Logical values are displayed, and may be entered, in any of four modes. The mode affects how a logical value is displayed on the screen after you save the definition. Logical values may be entered as decimal, octal, mnemonic, or hexadecimal regardless of the display mode. The current mode is displayed on the screen following the label `LOGICAL VALUE DISPLAY MODE`. It may be changed by typing c as the first character of a sequence while the cursor is in any of the `KEY STROKE` fields on the screen. When the miscellaneous keys screen is first invoked, the mode is hexadecimal. It cycles through all four modes when you press the c key. The four modes are:

- decimal

  In decimal mode, you enter logical values as decimal numbers. If the logical value is zero, an error will be displayed.

- octal

  In octal mode, you enter logical values as octal numbers (base 8). If the logical value is zero, an error will be displayed. Octal numbers are entered with a leading `0`.

- hexadecimal

  In hexadecimal mode, you enter logical values as hexadecimal (base 16) numbers. If the logical value is zero, an error will be displayed. Hexadecimal values are entered with a leading `0x`.

- mnemonic

  In mnemonic mode, you enter the mnemonic associated with any of the logical values stored in the `include` file `smkeys.h`. For example, if EXIT is entered into the `LOGICAL VALUE` field, the logical value of the EXIT key, hex 103, will be used. If an incorrect mnemonic is entered, an error will be displayed. For a list of valid mnemonics, press `?` key while the cursor is in a logical value field.

Entering the logical value as a mnemonic is preferable, as you are less likely to mistake the value you want. Using the numeric modes, it is possible to define logical key values other than those present in `smkeys.h`, but this should be done cautiously. You should avoid the range 100 hex through 1FF hex, which is reserved for future use by JYACC. Also, for portability's sake, the values should be small enough to fit in a two–byte integer, i.e., less than 65536 (10000 hex).

To define a key label or keytop for any key on this screen, press `k` with the cursor at the beginning of the key sequence. A small, borderless window will appear, bearing the word `KEYTOP:`. In the following field, you should type whatever appears on top of the key on your keyboard, using the < key to rubout mistakes. When done, press + to save the label, or – to discard it.

RETURNING TO THE MAIN MENU

To save changes made in this screen and return to the main menu, press the + key as the first character in a sequence while the cursor is in a `KEY STROKE` field. To discard the changes and return to the main menu, use the – key.

TEST KEYBOARD TRANSLATION FILE

This function allows you to test out your new key translation file. When "8" is selected from the main menu, the screen of Figure 11 is displayed. This screen has two fields labeled KEY STROKE and LOGICAL VALUE. You enter a keystroke (or sequence of keystrokes) that has been defined in another screen, and modkey will display the logical value of that key. The key or keys need only be pressed once, since the table is being tested for how it will behave when used in a real application.

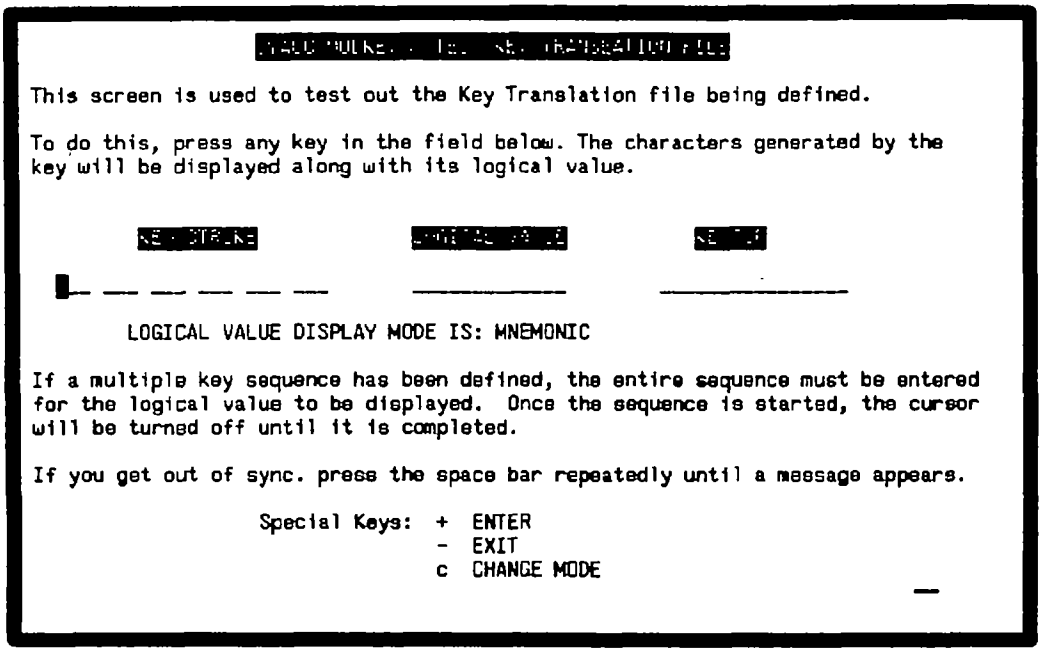


Figure 11: modkey Test Screen

If a key sequence forms only part of a previously specified sequence, modkey will wait for another key until a sequence is matched, or until it determines that no match is possible. In the latter case, the message Key not defined will appear.

The logical value can be displayed in any of the four modes (decimal, octal, hexadecimal, or mnemonic). To change modes, press c as the first character in a sequence. Help text can be obtained by pressing ?. To exit the screen and return to the main menu, use -.

ERRORS

Invalid entry.
Cause: You have typed a key that is not on the menu.
Corrective action: Check the instructions on the screen and try again.

Key sequence is too long.
Cause: You have typed more than six keys without repeating any.

Corrective action: Key sequences for translation may be at most six characters long. Choose a shorter sequence.

`Invalid first character.`
Cause: A multi-key sequence must begin with a control character.
Corrective action: Begin again, using a control character.

`Invalid mnemonic - press space for list`
Cause: In the miscellaneous keys screen, you have typed a character string for logical value that is not a logical key mnemonic.
Corrective action: Peruse the list, then correct the input.

`Invalid number - enter <decimal>, 0<octal> or 0x<hex>`
Cause: In the miscellaneous keys screen, you have typed a malformed numeric key code.
Corrective action: Correct the number, or use a mnemonic.

`Cannot create output file.`
Cause: An output file could not be created, due to lack of permission or perhaps disk space.
Corrective action: Correct the file system problem and retry the operation.

`Key sequence does not repeat.`
Cause: You have typed a key sequence that failed to repeat a string of six characters or less.
Corrective action: Retry the sequence, or use a shorter one.

`Cannot accept NUL as a key.`
Cause: The ASCII NUL character (binary 0) cannot be used in a key translation sequence, because it is used internally to mark the end of a sequence.
Corrective action: Use another key.

`Key previously defined as %s`
Cause: You have typed a key sequence that has already been assigned to another key.
Corrective action: Use a different key or sequence, or reassign the other.

WARNINGS

`Key overlaps with %s`
Cause: You have defined a sequence that is a substring of a previously defined sequence.
Action: You may define a sequence that is a substring of another sequence, but be sure that you have a timing interval set in the video file with the entry `KBD_DELAY`.

msg2bin

convert message files to binary

SYNOPSIS

```
msg2bin [-pv] [-e ext]  message-file...
msg2bin [-pv] [-o file] message-file...
```

OPTIONS

`-p`	Places each output file in the same directory as the corresponding input file.
`-v`	Prints the name of each message file as it is processed.
`-e`	Gives the output files the extension that follows the option letter, rather than the default `bin`. If `-o` is used, `-e` is ignored.
`-o`	Places all the output in a single file, whose name follows the option letter.

DESCRIPTION

The `msg2bin` utility converts ASCII message files to a binary format for use by **JAM** library routines.

The input[s] to this utility are text files containing named messages, either distributed by JYACC for use with the **JAM** library or defined by application programmers. For information about the format of ASCII message files, see the chapter on message files in the **JAM** *Configuration Guide*.

The message file and `msg2bin` utility provide three different services to application designers. First, the error messages displayed by **JAM** library functions may be translated from English to another language or altered to suit the taste of the application designer. Second, error messages for use by application routines may be collected in a message file and retrieved with the `msg_get` library function; this provides a centralized location for application messages and saves space. Finally, the standard library messages (and user messages) may be made memory–resident, to simplify and speed up the initialization procedure (at some added cost in memory). The `bin2c` utility converts the output of this utility to a source file suitable for inclusion in the application program.

Be aware that there is a maximum size to a message file that can be run through `msg2bin`. The sum over all messages of the message length + 2 must not exceed 65,529. If your message file is too long, separate it into two or more sections.

ERRORS

`File '%s' not found.`
Cause: An input file was missing or unreadable.

Corrective action: Check the spelling, presence, and permissions of the file in question.

`Unable to allocate memory.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

`Bad tag in line: %s`
Cause: The input file contained a system message tag unknown to the utility.
Corrective action: Refer to `smerror.h` for a list of tags, and correct the input.

`Missing '=' in line: %s`
Cause: The line in the message had no equal sign following the tag.
Corrective action: Correct the input and re–run the utility.

term2vid

create a video file from a terminfo or termcap entry.

SYNOPSIS

`term2vid [-f]` ***terminal-mnemonic***

OPTIONS

`-f` Overwrites existing file with the same name.

DESCRIPTION

This utility is only distributed on UNIX systems.

`term2vid` creates a rudimentary Screen Manager video file from information in the terminfo or termcap database. The video file produced by `term2vid` is limited by the description found in terminfo or termcap. You should test the video file which `term2vid` produces and expand it as necessary to include features not listed in terminfo and termcap. The documentation for your terminal should provide the necessary information. ***terminal-mnemonic*** is the name of the terminal type, the value of the system environment variable `TERM`, which is used by the C library function `tgetent` to access that database. The output file will be named after the mnemonic.

ERRORS

`No cursor position (cm, cup) for %s`
Cause: An absolute cursor positioning sequence is required for **JAM** to work, and the termcap or terminfo entry you are using does not contain one.
Corrective action: Construct the video file by hand, or update the entry and retry.

`Cannot find entry for %s`
Cause: The terminal mnemonic you have given is not in the termcap or terminfo database.
Corrective action: Check the spelling of the mnemonic.

`File %s already exists; use '-f' to overwrite.`
Cause: You have specified an existing output file.
Corrective action: Use the –f option to overwrite the file, or use a different name.

txt2form

converts text files to **JAM** screens

SYNOPSIS

`txt2form [-fv]` ***textfile screen*** `[`***rows columns***`]`

OPTIONS

`-f` Overwrites an existing output file.

`-v` Prints the name of each screen as it is being processed.

DESCRIPTION

This program converts ***textfile*** to a **JAM** screen, named ***screen***. It creates display data sections from the input text. It preserves blank space, and expands tabs to eight–character stops; other control characters are just copied to the output. Text that extends beyond the designated maximum output height or width is discarded; if the last two parameters are missing, the defaults are taken from the video file. The maximum value for ***rows*** and ***columns*** is 254.

`txt2form` puts no borders, fields, or display attributes in the output screen. However, underscores (or other, user–designated field definition characters) in the input are copied to the screen file; if you subsequently bring the screen up in `jxform` and compile it, those characters will be converted to fields.

ERRORS

`Warning: lines greater than %d will be truncated`
`Warning: columns greater than %d will be truncated`
Cause: Your input text file has data that reaches beyond the limits you have given (default 23 lines by 80 columns) for the screen.
Corrective action: Shrink the input, or enlarge the screen.

`Unable to create output file.`
Cause: An output file could not be created, due to lack of permission or perhaps disk space.
Corrective action: Correct the file system problem and retry the operation.

var2bin

convert files of setup variables to binary

SYNOPSIS

```
var2bin [-pv] [-e ext] setupfile...
```

OPTIONS

`-p`	Puts output file in same directory as input file.
`-v`	Prints the name of each setup file as it is converted.
`-e`	Appends ***ext*** to the name of each output file. The default is `bin`.

DESCRIPTION

This utility converts files of setup variables (i.e., `SMVARS` and `SMSETUP`) to binary format for use with **JAM** library routines. See the chapter on setup files in the **JAM** *Configuration Guide* for a full description of how to prepare the ASCII file.

ERRORS

`Error opening %s.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the file in question.

`Missing '='.`
Cause: The input line indicated did not contain an equal sign after the setup variable name.
Corrective action: Insert the equal sign and run var2bin again.

`%s is an invalid name.`
Cause: The indicated line did not begin with a setup variable name.
Corrective action: Refer to the *Configuration Guide* for a list of variable names, correct the input, and re–run the utility.

`%s may not be qualified by terminal type.`
Cause: You have attached a terminal type list to a variable which does not support one.

Corrective action: Remove the list. You can achieve the desired effect by creating different setup files, and attaching a terminal list to the `SMSETUP` variable.

`Unable to set given values.`
`%s conflicts with a previous parameter.`
`%s is an invalid parameter.`

Cause: A keyword in the input is misspelled or misplaced, or conflicts with an earlier keyword.
Corrective action: Check the keywords listed in the manual, correct the input, and run the utility again.

`Error reading smvars or setup file.`
Cause: The utility incurred an I/O error while processing the file named in the message.
Corrective action: Retry the operation.

`Unable to allocate memory.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

`At least one file name is required.`
Cause: You have failed to give an input file name.
Corrective action: Retype the command, supplying the file name.

`Entry size %d is too large.`
`String size %d is too large.`
Cause: The indicated right–hand side is too long.
Corrective action: Reduce the size of the entry.

vid2bin

convert video files to binary

SYNOPSIS

```
vid2bin [-pv] [-e ext] video-file
```

OPTIONS

`-p`	Puts output file in same directory as input file.
`-v`	Prints the name of each setup file as it is converted.
`-e`	Appends ***ext*** to the name of each output file. The default is `bin`.

DESCRIPTION

The `vid2bin` utility converts an ASCII video file to binary format for use by applications with the JAM library routines. The video files themselves must be created with a text editor, according to the rules listed in the video manual.

video-file is an ASCII video file. Customarily, it is an abbreviation for the name of the terminal for which the ASCII video file has been constructed followed by the suffix `vid`, for example `sunvid` for a terminal, or `colvid` for a color monitor. When searching for the file, `vid2bin` first tries the mnemonic, then the mnemonic followed by `vid`. The output file is named after the input file, with the extension `bin` or the extension `ext` specified with `-e`.

In errors are encountered during the conversion, JAM will display up to 10 error messages. No output file will be created.

To make a video file memory-resident, run the `bin2c` utility on the output of `vid2bin`, compile the resulting program source file, link it with your application, and call the library routine `vinit`. For information about the format of the ASCII video file, refer to the video manual and the *Programmer's Guide*.

ERRORS

`Neither %s nor %s found.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the file in question.

`A cursor positioning sequence is required.`
`An erase display sequence is required.`

Cause: These two entries are required in all video files.
Corrective action: Determine what your terminal uses to perform these two operations, and enter them in the video file; then run the utility again.

`Unable to allocate memory.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: None.

`Error writing to file '%s'.`
Cause: The utility incurred an I/O error while processing the file named in the message.
Corrective action: Retry the operation.

```
Invalid entry: '%s'.
Entry missing '=': '%s'.
```
Cause: The input line in the message does not begin with a video keyword and an equal sign.
Corrective action: Correct the input and re–run the utility. You may have forgotten to place a backslash at the end of a line that continues onto the next one.

```
Invalid attribute list : '%s'.
Invalid color specification : '%s'.
Invalid graphics character specification (%s):'%s'.
Invalid border information (%s):'%s'.
Invalid graphics type : '%s'.
Invalid label parameter : '%s'.%s
Invalid cursor flags specification : '%s'.
```
Cause: You have misspelled or misplaced keywords in the input line in the message.
Corrective action: Correct the input, referring to the *Configuration Guide*, and run vid2bin again.

INDEX

Symbols

A

B

C

D

E

F

I

J

K

L

M

O

P

R

S

T

U

V

Glossary

Glossary

The following terms are used throughout this manual in describing JAM. Some familiarity with them will help you learn JAM faster.

active screen — The *screen* at the top of the *window stack*. This is the screen that the user may interact with. If there is a cursor, it will appear on this screen.

alternative scrolling — See *custom scrolling*.

application executable — The executable version of an application that is given to the user. It does not allow access to the JAM Data Dictionary, Screen, or Keyset Editors.

application mode — The start–up mode in an *authoring* session. In this mode, the developer may test a complete application, including individual screens and inter–screen links. Any of the editing modes may also be entered from application mode. Application mode may be customized to enable testing of application–specific, third generation language (e.g. C) code. (Contrast with *TEST mode* of the *Screen Editor* where screens can be tested, but inter–screen links cannot.)

array — A field comprising one or more occurrences. Each occurrence may contain data. The on–screen portion of an array contains one or more field *elements*, where each field element is identified by a field number that is unique on the screen. An array may contain more *occurrences* than it has on–screen *elements*. In that case it is called a *scrolling array*.

attached functions — This term is no longer used as of JAM release 5. See field functions, group functions, and screen functions.

attributes — The description of how a particular section of the screen will look, in terms of color, brightness, underline, blinking etc. Each *field* or *display text* area has a set of attributes. These are also called *display attributes*.

authoring — The act of using the `jxform` utility to create a JAM application.

authoring executable — An executable version of an application that provides access to JAM's Data Dictionary, Keyset, and Screen Editors, as well as to developer–written hook functions. It is created by

	linking these hook functions to the authoring libraries to create a new `jxform` authoring tool. An authoring executable can be useful to test the functionality of an entire application while allowing instant access to the Screen, Data Dictionary, and Keyset Editors.
authoring environment	The tools used to create and test **JAM** screens and links, the authoring environment includes ***application mode***, the ***Screen Editor***, ***Data Dictionary Editor***, and optionally, the ***Keyset Editor***. (Contrast with ***runtime environment***.)
authoring tool	See `jxform`.
block mode terminal	A terminal that collects user input and transmits a set of new or changed data to the host computer in a block, as opposed to a terminal that transmits each keystroke separately.
caret function	Term no longer used as of **JAM** release 5. See ***control function***.
character edit	A restriction placed by the developer on the type of characters (digits, numerics, letters, and so on) that may be entered in a ***field***. These are enforced by **JAM** on the user's data entry in the field.
checklist	A group of fields, zero or more of which may be selected by the user.
control field	See ***menu control field***.
control string	Control strings can be thought of as the links that hold a **JAM** application together. A control string defines the action to take place when function key and menu selection ***events*** occur. Possible actions include: displaying a form or window, calling a developer–written function, or executing a system command.
control function	A developer–defined function that is called via a ***control string***. These were formerly known as *caret functions* and *invoked functions* prior to **JAM** release 5.
cursor	A marker on the display that indicates where any text that is typed will appear. It is usually a blinking reverse video block or a blinking underline character.
custom scrolling	An alternative to **JAM**'s default (memory–based) mechanism for storing off–screen array occurrences. Also called ***alternative scrolling*** or ***disk–based scrolling***.

data dictionary — A repository of information about fields and groups. The data dictionary serves two major functions: (1) during the authoring process it provides a means of propagating field attributes from screen to screen, (2) when an application is run, the data dictionary (often abbreviated DD) is used to create the ***Local Data Block***.

Data Dictionary Editor — A tool used to create and modify the ***data dictionary***. It is part of the ***authoring environment***.

developer — One who builds **JAM** applications.

disk–based scrolling — See ***custom scrolling***.

display — The physical screen of a terminal or monitor. (Contrast with a **JAM** ***screen***, which is a software object.)

display text — Those portions of a **JAM** screen that typically are not altered by the user or the program at runtime. Display text consists of text and graphical characters that are not in ***fields*** and that are not part of the screen border.

DRAW mode — One of four editing modes in the ***Screen Editor***, DRAW mode supports entry of display text, creation of fields, and entry of initial data into fields. (Contrast with ***line–drawing mode***, ***TEST mode***, and ***select mode***.)

element — An on–screen field in an array. (Contrast with ***occurrence***.)

embedded punctuation — Punctuation characters embedded (as initial data) in a digits only, alpha, or alphanumeric field. The positions occupied by the embedded punctuation are skipped during data entry, thereby providing a mechanism for easily entering structured data such as telephone numbers.

event — An event is simply something that has happened. Most events are associated with a user interacting with an input device by pressing a function key or making a menu selection. Events often determine the flow of an application.

executive — See ***JAM Executive***.

field — A part of a screen typically populated at runtime. This includes areas in which the user enters data or makes a menu selection, or the program displays variable output. Each field on a screen is uniquely identified by a field number. A field may also have a name. It must have a name if it is to be entered in the ***data dictionary***. (Contrast with ***display text***.)

field attachments — Field name, next field specification, previous field specification, item selection screen, table lookup screen, status text, memo text, and validation function are all field attachments. Help screens, entry functions and exit functions may also be attached to fields.

field edits — Restrictions placed on user data that can be entered into a ***field***. Field edits will either (1) change the user's input to match the field edit (for example, right justify text or capitalize lower case entries), or (2) reject the user's entry entirely when the field is ***validated*** (as with check–digit and range checking field edits).

field function — A function associated with field events (ie. — field entry, validation, and exit).

field number — A unique number automatically assigned to a **JAM** screen ***field*** based on its position on the screen. Fields are numbered from left to right beginning at the top line of the screen and ending at the bottom line. Thus, the left–most field on the top line of the screen would have field number 1, and the right–most field in the bottom line would have the highest field number on the screen.

field validation — The action of checking that the contents of a field meet the criteria described in the field's validation function, as well as in the ***field's edits***. Field validation generally occurs when a field is exited with a tab, or when the `TRANSMIT` key is pressed.

form — One of the two ways a developer may select to display a **JAM** ***screen*** within an application. When displayed as a form, a screen replaces the current form and closes all open ***windows***. Only one form may be open at a time. (Contrast with a screen displayed as a ***window***, which preserves any existing forms or windows.)

form stack — An ordered list of the names of screens that have been displayed as forms in the application. The form stack is maintained by the **JAM** Executive. When a screen is displayed as a form, it is added to the top of the form stack. When a form is closed, it is popped from the form stack and the previously displayed form is opened.

formaker — A term no longer used as of **JAM** release 5. See ***Screen Manager*** and ***Screen Editor***.

function key — A key with a function other than data entry. Function keys are normally associated with control strings that specify such actions as form and window display. **JAM** function keys are defined as logical keys, and the developer decides which physical key to map each function to. This feature enables **JAM** applications to be terminal independent.

function list — A list that maps function names to function memory addresses. A function list is needed if certain developer–written, compiled code (hook functions) will be used within the application.

group — One or more fields that are part of a radio button or checklist. Groups have characteristics such as entry/exit/validation functions and tabbing order.

group functions — Functions associated with group events (group entry, exit and validation).

help screen — An informational screen that may be attached to a **JAM** ***screen*** or ***field***. A help screen is displayed when the `HELP` or `SCREENHELP` key is pressed.

hook functions — Developer–written functions that are linked into an application. They may be called by **JAM** during field, group, or screen entry/exit/validation, when a function key is pressed, when a menu item is selected, or at various other times.

infinite loop — See ***loop***.

invoked function — Term no longer used as of **JAM** release 5. See ***control function***.

JAM Executive — The portion of **JAM** that manages the order of screen presentation. The **JAM** Executive performs this task by processing all ***control strings***, and then directing the ***Screen Manager*** to display the appropriate screen in the manner specified. The **JAM** Executive processes control strings in response to the Screen Manager's reports of function key and menu field selection events. Developers may also write custom executives.

JPL module — A collection of one or more ***JPL procedures*** that are stored together.

JPL procedure — A subroutine written in the JPL programming language.

`jxform` — The program that contains ***application mode***, and the ***Screen, Data Dictionary, and Keyset Editors***. It is also generally the

operating system command entered to start an authoring session. `jxform` is also referred to as the ***authoring tool***.

key translation table — A file that contains the mapping between **JAM** ***logical keys*** and the physical keys on a terminal. You may construct or change the key translation table with a text editor, or via the `modkey` utility.

keyset — A mapping between a terminal's ***soft keys*** and their logical value. A keyset may contain several rows of soft key definitions. Each definition consists of a soft key label, an optional display attribute, and a logical value for the soft key.

Keyset Editor — The tool used to create and modify ***keysets***. It is an optional part of the ***authoring environment***.

LDB — See ***local data block***.

line–drawing mode — One of four editing modes in the Screen Editor. Line–drawing mode allows the developer to draw lines and boxes using the cursor as a pen. Line–drawing mode can be entered from the ***Screen Editor***. (Contrast with ***DRAW mode***, ***TEST mode*** and ***select mode***).

local data block — A repository of ***data dictionary*** entries and values that is automatically created at run–time. The local data block transmits those values to fields or groups on screens that share the name of fields or groups in the data dictionary.

logical key — A device independent mnemonic for a pre–defined **JAM** function. These logical functions have names such as `TRANSMIT`, `EXIT`, and `MENU TOGGLE`. Logical keys are mapped to physical keys via the keyboard translation table.

loop — See ***infinite loop***.

menu — The set of fields on a screen that have the menu edit. A user selects an item from a menu field to tell the **JAM** application what action to perform next.

menu control field — The field directly to the right of a ***menu*** field. It contains a control string to be executed when the menu item is selected. Menu control fields are usually non–display fields.

message file — A configuration file that maps message text to message identifiers. All **JAM** messages are stored in the message file to enable customization (e.g. — for international use). Developers may use the message file for application messages as well.

The message file also contains global date, time, and currency formats.

occurrence — A component of an array. The number of occurrences may be larger than the number of on–screen ***elements***. Thus an occurrence may be on– or off–screen. (Contrast with ***element***).

parallel array — A term no longer used as of **JAM** release 5. See ***synchronized array.***

radio buttons — A ***group*** of fields, only one of which may be chosen at a time (like the buttons on a a car radio). Making a selection automatically de–selects the previous selection.

runtime environment — The environment in which a user runs a **JAM** application. This environment acts like the ***authoring environment***, but it does not contain the ***Screen, Data Dictionary, and Keyset Editors***.

scope — A number between 0 and 9 that is associated with a ***data dictionary*** entry. A scope of zero prevents an entry from being included in the ***local data block*** at runtime. Entries with the same scope in the local data block can be erased and re–initialized as a group.

screen — A term used to refer generally to ***forms*** and ***windows***.

screen binary — The representation of a screen that is created by the ***Screen Editor***. The Screen Editor stores the screen binary into a file (typically called ***screenname***.`jam`). The screen binary contains all of the information the ***Screen Manager*** needs at runtime to display the screen and to manage user interaction with the screen. The screen binary also contains the control strings that the ***executive*** uses to determine the order of screen presentation.

Screen Editor — The Screen Editor supports the creation and testing of individual **JAM** screens. It is called from within the ***authoring environment.***

screen function — A function associated with a screen event (entry or exit). A screen entry function is called when a screen is initially displayed or when it is subsequently activated (exposed). A screen exit function is called when a screen is closed or deactivated (hidden).

Screen Manager — The **JAM** library functions that manage the display of individual screens, and user interactions with those screens. The

Screen Manager handles ***events*** associated with a single screen, such as field entry, field exit, group entry, group exit, screen entry, screen exit, and menu field selection events. The Screen Manager does not handle the order in which screens are displayed or the action of any control strings. Instead it notifies the **JAM** ***executive*** of such activities.

scrolling array — An array that has more ***occurrences*** than it has on–screen ***elements***. Thus an array may contain a list of data longer than the available screen space. The user can scroll the array to bring off–screen occurrences on to the screen.

select mode — Select mode may be entered from ***DRAW mode*** or ***TEST mode*** in the ***Screen Editor***. In select mode, the developer may select a block of display text and fields that can then be positioned or edited as a single object.

setup file — At runtime, **JAM** may use setup files to learn about an application's configuration. The environment variables placed in these files tell **JAM** how the application should look, how the hardware is configured, and where important system files are located. Configuration files can be used to establish installation–wide configuration information and application–specific configuration information.

shifting field — A field that may contain more data than fits in the horizontal area apportioned to it on the screen. Using the arrow keys, the user can shift the data in the field horizontally.

sibling windows — ***Windows*** that are at the same level as each other. Siblings can be opened and activated without closing the calling window. In some windowing environments, these are called *non–modal* windows. The viewport key allows the user to select among (activate) any open sibling windows. (Contrast with ***stacked windows***).

soft key — A logical key whose action can change during the course of an application, depending on the context in which it is called. **JAM** uses a ***keyset*** to determine the action that a particular soft key will take. The keyset also contains screen labels that inform the user of the action of the soft key. Soft keys can be simulated on terminals that do not provide hardware support for them.

stacked windows — ***Windows*** that appear on the display in a layered fashion. Once a stacked window has been opened, it must be closed

before the user can access any underlying screens. In some windowing environments, these are called *modal* windows. (Contrast with ***sibling windows***.)

status line The line of the terminal display—usually the bottom line—that may be reserved for displaying error and status messages.

status text A text string that appears on the terminal's ***status line*** at run-time, whenever the cursor enters a particular field. The status text is attached to the field.

synchronized arrays Two or more ***scrolling arrays*** on a screen whose data scroll together. Synchronized arrays need not be adjacent to each other on the screen, but they must contain the same number of total ***occurrences*** and on-screen ***elements***.

system date/time The current date and time stored in the computer. **JAM** date/time fields can be automatically initialized to the system date/time.

TEST mode One of four editing modes in the ***Screen Editor***, TEST mode permits the developer to test individual screens during a screen editing session. Since **JAM** ***control strings*** are not operational in TEST mode, only functions and edits relating to a single screen may be tested here. (Contrast with ***application mode*** where control flow can be tested, and with ***line drawing mode***, ***select mode***, and ***DRAW mode***.)

user The end-user of a **JAM** application. (Contrast with ***developer***)

validation The process of checking user data entry against the ***field edits*** imposed by the developer. Validation generally occurs when a field is exited via the TAB key. Validation generally occurs for all fields on a screen when the XMIT key is struck.

video file A file that tells **JAM** how to use the capabilities of a specific terminal. **JAM** includes a set of video files for many different terminals. Optionally, the developer may define a custom video file using the instructions in the **JAM** *Configuration Guide*.

viewport The mechanism through which the user views all or part of a ***virtual screen***. Each **JAM** screen has its own viewport. The viewport key allows the user to move, resize, and scroll the viewport.

virtual screen An entire screen, including the visible and non-visible portions of it in a ***viewport***. Unless discussing screens in the con-

text of viewports, the simpler term ***screen*** is generally used instead. Virtual screens may be as large as 254 rows by 254 columns.

window — One of the two ways that **JAM** can display a screen in an application. Unlike a screen displayed as a ***form***, a screen displayed as a window overlays but preserves any screens beneath it. The open window becomes the ***active screen*** and the image hidden by the open window is saved, to be restored and become active again when the open window is closed. Windows may be ***stacked*** or ***sibling***. (Contrast with a ***form***).

window stack — A list that is kept internally by **JAM** that allows it to remember the order in which ***windows*** were opened or rearranged with the viewport key. When a window is closed, it is *popped* off the window stack, and the previous window is re–displayed.

word wrap — An optional attribute that may be given to an ***array*** which causes the ***occurrences*** in the array to be treated like lines of text. **JAM** will not split lines in the middle of words. When characters are deleted from an occurrence, **JAM** automatically fills the line with words from the next occurrence in the array. When characters are inserted, **JAM** will wrap any text which is too lengthy for the current occurrence to the next occurrence in the array.

working pen — In the ***Screen Editor***, the developer may specify a working pen to describe the display ***attributes*** to be used for anything that is added to the screen. To add ***display text*** or ***fields*** with different attributes, simply change the attributes of the working pen.

zoom — Any field that shifts or scrolls may be viewed and edited in full using the ZOOM key. A pop–up window will appear that contains the entire on–screen and off–screen contents of the field, to the extent permitted by the physical display.

Upgrade Guide

JAM Upgrade Guide

Release 4 to Release 5

Introduction

You should probably skim through the Release 5 manual before upgrading to let you get an idea of some of the new features that we've added, and where to look to find information from the manual. By following the step by step instructions in this document you will be up and running quickly and easily.

Step by Step Instructions

1. **Back up your Release 4 Application.**
2. **Install JAM Release 5.**

   Go to a different location on your disk than where you keep Release 4. We do not recommend removing Release 4 until Release 5 is running your application successfully.

   Follow the step by step Installation notes that came with the Release 5 media.
3. **Update your environment.**

   Your operating system will have to know where you have just installed the **JAM** 5 libraries and executables. **JAM** will have to know where you have put your message and other configuration files.
4. **Convert Screens from Release 4 to Release 5 Format.**

   You can easily convert Release 4 screens to Release 5 by running the `f4to5` utility. Check the *Utilities Guide* for useful information on how to use this utility. For example, to upgrade your existing Release 4 screens named `screen1.jam` and `screen2.jam`, create a Release 5 application directory, then go to that directory and issue the following command:

   ```
   f4to5 -v rel4dir/screen1.jam rel4dir/screen2.jam
   ```

   Depending on your operating system, you may need to use a different method of referencing the screens in the Release 4 directory. The `-v` flag will cause the computer, upon successful completion, to list `screen1.jam` and `screen2.jam`.

   NOTE: Your new Release 5 screens will be almost identical except that your DATE/TIME fields and CURRENCY FORMAT edits will be in Release 5 format. The con-

version utility will convert the formats from Release 4 to Release 5. For example, the mnemonic used for a 4 digit year was YYYY. The conversion utility will convert YYYY to the Release 5 format, YR4. Note that if your application accessed the Release 4 mnemonics with `sm_edit_ptr`, then you may need to change application code.

To upgrade memory resident screens, first get the disk resident screens that you created the memory resident screens from originally. Use `f4to5` to convert them from Release 4 to 5. Then create memory resident screens from the upgraded disk resident screens.

5. **Convert your data dictionary from Release 4 to Release 5 format.**

   Your data dictionary files must be updated with the `dd4to5` utility. Check the *Utilities Guide* for useful information on how to use this utility.

   One method of updating your existing Release 4 data dictionary is to go to your Release 5 application directory and then issue the following command (assuming that the name of the dictionary is `data.dic`):

   ```
   dd4to5 -f rel4dir/data.dic
   ```

   Depending on your operating system, you may need to use a different method of referencing the data dictionary in the Release 4 directory.

6. **Eliminate Use of Release 3 Library Routines**

   Routines that were supported in Release 4 for compatibility with Release 3 are no longer supported. All usage of Release 3 routines must be updated to use Release 5 routines.

JAM 3		**JAM 5**
`jm_ch_form_atts`	=>	`sm_option`
`ldb_e_get`	=>	`sm_i_fptr`
`ldb_e_put`	=>	`sm_i_putfield`
`ldb_get`	=>	`sm_n_fptr`
`ldb_init`	=>	`sm_ldb_init`
`ldb_l_clear`	=>	`sm_lclear`
`ldb_l_reset`	=>	`sm_lreset`
`ldb_merge`	=>	`sm_allget`
`ldb_put`	=>	`sm_n_putfield`
`ldb_stat`	=>	`sm_n_length; sm_n_max_occur`
`ldb_store`	=>	`sm_lstore`
`ldb_t_reset`	=>	`sm_lreset`
`ldb_t_clear`	=>	`sm_lclear`
`sm_doesshift`	=>	`sm_t_shift`
`sm_g_item`	=>	`sm_o_gofield`

```
sm_hasscroll        =>    sm_t_scroll
sm_inoff            =>    sm_ind_set
sm_inon             =>    sm_ind_set
sm_n_g_item         =>    sm_i_gofield
sm_n_num_items      =>    sm_n_num_occurs
sm_n_read_item      =>    sm_i_fptr
sm_n_w_item         =>    sm_i_putfield
sm_num_items        =>    sm_num_occurs
sm_read_item        =>    sm_o_fptr
sm_w_item           =>    sm_o_putfield
```

7. **Eliminate References To Global Variables**

If you were a little more adventurous than most and used some of the **JAM** internals that you found in the include files, then read the following paragraph. If `sm_do_not_display` or `sm_fldptrs` seem familiar to you, then you are in this category.

You should now take the opportunity to remove those unsupported globals from your code. They may still work, but it is a good idea to use the new SUPPORTED functions that we have added. These functions are `sm_iset`, `sm_pset`, `sm_inquire`, `sm_pinquire`, and `sm_bitops` (`sm_bitops` was also documented in release 4). Check the **JAM** *Programmer's Guide* for the descriptions of these functions.

8. **Update your binary configuration files.**

Go to the Configuration directory. If you had made any changes in these files in Release 4, you will have to duplicate the changes in the Release 5 versions of these files. In particular, note that:

- KSET in the video file takes different arguments. If you modified KSET, see "video file" in the *Configuration Guide* for further information.
- Status line definitions are no longer allowed in the video file. All status line definitions are now in the message file.

Be especially careful when modifying your message file. Copy your user messages into the message file. If you changed any **JAM** messages make sure that you are changing the same messages in Release 5. Some message mnemonics have been added, deleted and, in a few cases, subtly changed their meaning.

Run the following utilities on your new Release 5 text configuration files.

`key2bin`	Updates key files
`msg2bin`	Updates message file
`var2bin`	Updates `SMVARS` and `SMSETUP` files
`vid2bin`	Updates video files

See the *Utilities Guide* for information on how run these utilities. REMINDER: Set the environment variable `SMVARS` to point to the file (usually `smvars.bin`) that you ran through `var2bin`.

Note that the following **JAM** 4 `var2bin` mnemonics are included for backward compatability.

```
SMCHEMSGATT
SMCHFORMATTS
SMCHQMSGATT
SMCHSTEXTATT
SMCHUMSGATT
SMDWOPTIONS
SMEROPTIONS
SMFCASE
SMFEXTENSION
SMINDSET
SMMPOPTIONS
SMMPSTRING
SMOKOPTIONS
SMUSEEXT
SMZMOPTIONS
```

9. Check your JPL procedures.

'&' and '|' cannot be used as logical operators. They are now bitwise operators. This should not cause a problem in most cases.

Colon syntax is not supported for specifying occurrences.

10. Compile your JAM application.

See your installation notes for the locations and names of JAM libraries.

11. Don't PANIC.

If nothing seems to work, check you environment variables, and make sure your operating system is using the latest libraries and utilities.

Make sure you converted your latest version of your application. If it didn't work in Release 4, then it probably won't work in Release 5.

If `jxform` and other utilities print out a number enclosed in angle brackets, such as `<4-687>`, then check your message file; **JAM** can't find or read it.

Make sure that you are using utilities the way they work in Release 5 not the way you remember them from Release 4.

Check the manual for the proper usage of utilities. You may have missed the description of a flag that you needed or reversed the order of some parameters. In addition, the examples may be helpful.

Check the sample applications that come with **JAM**.

For Your Information

Once you've upgraded your application from release 4 to 5, you'll want to be aware of the following information.

Parameter windows are obsolete with the advent of colon preprocessing. They are not supported in release 5.02 unless you add the following lines to the application's main routine (`jmain` or `jxmain`).

```
extern int sm_param_windows;

sm_param_windows = 1;
```

Colon preprocessor variables should be used in place of parameter windows to ensure compatability with future releases. The removal of prarmeter windows enables control strings that contain a percent character (%), as in a percent escape.

The content of `smlint.h` has been moved to `smproto.h`.

`smsetup.h` now contains various setup parameters that may be altered within the SMVARS and SMSETUP files or at runtime with `sm_option`. See setup file in the *Configuration Guide* and `sm_option` in the *Programmer's Guide*.

The contents of `smdefs.h` have been divided into the following files:

- `smmach.h` platform specifications
- `sminstfn.h` function list
- `smmouse.h` mouse support
- `smattrib.h` display attributes
- `smumisc.h` argument definitions for miscellaneous functions
- `smsetup.h` setup variables
- `smvalids.h` validation bits
- `smedits.h` special edits
- `smproto.h` function list

The source code of `jxmain.c` and `jmain.c` has been altered. You no longer need to comment-out code that initializes **JAM** subsystems not applicable to your application. Instead, a list of macros in `jxmain.c` and `jmain.c` govern the inclusion of optional code. They come set to `0` for not included. To enable a subsystem, simply set the appropriate macro to `1`.

Screens whose bottoms were cut off, because they are too big for the terminal they are running on, they will now be displayed through a viewport with scroll bars across the bottom to indicate off-screen data.

The logical key RETURN behaves differently within menus. In release 4 RETURN moved the cursor to the first menu field on the next line. In release 5 RETURN, like XMIT (transmit), selects the current menu field.

The following functions are supported in Release 5 only for compatibility with Release 4. The Release 5 functions should be used to ensure compatibility with future releases.

JAM 4		**JAM** 5
sm_ch_emsgatt	=>	sm_option
sm_ch_form_atts	=>	sm_option
sm_ch_qmsgatt	=>	sm_option
sm_ch_stextatt	=>	sm_option
sm_choice	=>	sm_input
sm_cl_everyfield	=>	sm_unprot
sm_dw_options	=>	sm_option
sm_er_options	=>	sm_option
sm_fcase	=>	sm_option
sm_fextension	=>	sm_pset
sm_inbusiness	=>	sm_inquire
sm_menu_proc	=>	sm_input
sm_mp_options	=>	sm_option
sm_mp_string	=>	sm_option
sm_ok_options	=>	sm_option
sm_openkeybd	=>	sm_input
sm_plcall	=>	sm_jplcall
sm_sdate	=>	sm_sdtime
sm_stime	=>	sm_sdtime
sm_smsetup	=>	sm_option
sm_unsetup	=>	sm_option
sm_zm_options	=>	sm_option

These functions are new to Release 5:

- sm_bkrect — Set background color of rectangle.
- sm_c_keyset — Close a keyset.
- sm_d_keyset — Display a memory–resident keyset.
- sm_deselect — Deselect a checklist occurrence.
- sm_fi_open — Find a file and open it in binary read only mode.
- sm_fi_path — Return the path name of a file.
- sm_finquire — Obtain information about a field.
- sm_ftog — Convert a field number and occurrence into a group name and index.
- sm_ftype — Get the data type and precision of a field.
- sm_getjctrl — Get control string associated with a key.
- sm_gp_inquire — Obtain information about a group.
- sm_gtof — Convert a group name and index into a field number and occurrence.

- `sm_gval` — Force group validation.
- `sm_input` — Open the keyboard for data entry and menu selection. Replaces version 4 `sm_choice`, `sm_menu_proc`, and `sm_open-keybd`.
- `sm_inquire` — Obtain value of a global integer variable.
- `sm_is_no` — Test field for no.
- `sm_iset` — Change value of integer global variable.
- `sm_isselected` — Determine whether a radio button or checklist occurrence has been selected.
- `sm_issv` — Determine if a screen in the saved list.
- `sm_jplload` — Execute the JPL `load` verb.
- `sm_jplpublic` — Execute the JPL `public` verb.
- `sm_jplunload` — Execute the JPL `unload` verb.
- `sm_keyset` — Open a keyset
- `sm_keyoption` — Set key option.
- `sm_kscscope` — Query current keyset scope.
- `sm_ksinq` — Inquire about key set information.
- `ks_label` — Set a soft key label and attribute.
- `sm_ksoff` — Turn off key labels.
- `sm_kson` — Turn on key labels.
- `sm_mnutogl` — Switch between menu mode and data–entry mode on a dual–purpose screen.
- `sm_name` — Obtain field name given field number. `sm_null` Test if field is null.
- `sm_option` — Set a screen manager option. Replaces the following version 4.0 functions:
  - `sm_ch_emsgatt`
  - `sm_ch_form_ats`
  - `sm_ch_qmsgatt`
  - `sm_ch_umsgatt`
  - `sm_dw_options`
  - `sm_er_options`
  - `sm_fcase`
  - `sm_fextension`
  - `sm_ind_set`
  - `sm_mp_options`
  - `sm_mp_string`
  - `sm_ok_options`
  - `sm_stextatt`
  - `sm_zm_options`
- `sm_pinquire` — Obtain value of a global string.
- `sm_pset` — Modify value of a global string.

- `sm_sdtime` Get formatted system date and time.
- `sm_select` Select a checklist or radio button occurrence.
- `sm_shrink_to_fit` Remove trailing empty array elements and shrink screen.
- `sm_sibling` Define the current window as being or not being a sibling window.
- `sm_skinq` Obtain soft key information by position.
- `sm_skmark` Mark or unmark a softkey label by position.
- `sm_skset` Set characteristics of a soft key by position.
- `sm_skvinq` Obtain soft key information by value.
- `sm_skvmark` Mark a soft key by value.
- `sm_skvset` Set characteristics of a soft key by value.
- `sm_soption` Set a string option.
- `sm_submenu_close` Close the current submenu.
- `sm_svscreen` Register a list of screens on the save list.
- `sm_udtime` Format user–supplied date and time.
- `sm_unsvscreen` Remove screens from the save list.
- `sm_viewport` Modify viewport size and offset.
- `sm_wcount` Obtain number of currently displayed windows.
- `sm_winsize` Allow end–user to interactively move and resize a window.
- `sm_wrotate` Rotate the display of sibling windows.

Master Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Addendum

for Updates to JAM Release 5.03 Volume 1

for Stratus

Part Number R330–01A

August 3, 1992

Note of Explanation

This addendum describes new features in release 5.03 of **JAM**. This addendum is for Volume 1 of the documentation set. There is a separate addendum for Volume 2. Descriptions of the features are broken into sections based on the parts of the manual that they affect.

Configuration Guide

Page 33: `read.me` File in VOS Distirbutions

The documentation refers to the `read.me` file. This file does not exist on VOS distributions. Refer to your installation notes instead.

Page 39: VOS Pathnames

Under VOS, the pathnames specified for variables like SMKEY and SMMSGS should be a standard VOS pathname followed by a >, as in: `>directory_name>`

Page 39: Choosing an Editor in JPL

A new setup variable has been added that allows the developer to select an editor to use when entering text into a JPL module in the screen editor. The variable is called SMEDITOR, and may be set in the setup or smvars file as follows:

```
SMEDITOR= vi
```

To invoke the editor from a JPL module, press the PF5 function key. To set the variable at runtime, use the library routine `sm_soption` with the argument `SO_EDITOR`.

Page 96: Color of Shift/Scroll Indicators

Shift/Scroll indicators are white, unless the screen background color is yellow, white or cyan, in which case the indicators are black. Formerly, the indicators were always white.

JAM

VOLUME II

- JPL Guide
- Programmer's Guide

The Composer for Sophisticated Applications.

JPL Guide

TABLE OF CONTENTS

Chapter 1
Introduction

JPL is the JYACC Procedural Language. Since it is an interpreted language, it helps you rapidly develop **JAM**® applications. With JPL you can write and execute functions without leaving the authoring session. Unless you call application code or library functions with JPL, you do not need to compile your **JAM** application to test these procedures.

You might write a JPL routine, for example, which displays a message to the end–user based on his entry in a data field. You might write a procedure which creates a string expression from the contents of data entry fields and displays the expression on other screens. Screen, field, group, and control string functions may all be written in JPL. In these functions you can perform numeric calculations, test conditions, use loops, and call other JPL routines. As mentioned, you may also call application code and library functions in a compiled **JAM** application.

JAM provides utilities to improve the performance time of JPL procedures. For example, you can eliminate runtime JPL compilation by storing procedures in binary files. To protect the end–user from accidentally changing procedures, you may store binary files in a library. Chapter 9 – Performance Considerations, explains how to use the **JAM** utilities to make these improvements.

While programming experience will help you use JPL, it is not a prerequisite. We assume, however, that you are familiar with the **JAM** Screen Editor.

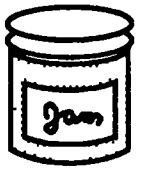
1.1
CONVENTIONS USED

To make this guide easier to use, we use the following conventions. Familiarity with them will help you understand and use the material more quickly.

- `literal` We use this font for words which you will type verbatim. In particular, we use this font for all our examples. In addition, when we name a JPL command, JAM utility, or JAM library function we use this font to distinguish it from the standard text.
- ***italics*** We use bold italics to show where screen, file, and variable names should appear. You should replace these with the appropriate names in your applications.
- ***{x}*** In this notation, the brackets indicate that ***x*** is an optional element. The brackets should not be typed.
- ***x...*** Ellipses indicate that the element ***x*** may be repeated one or more times.

1.2
ABOUT THIS DOCUMENT

- The next chapter is a "quick start" to JPL. We guide you through the creation of a JPL procedure in a sample application.
- In Chapter 3 we explain the tasks of writing, storing, and calling JPL procedures.
- In Chapter 4 we describe JPL variables, and in Chapter 5 we explain the related topic, colon expansion.
- In Chapter 6 we cover data types, operators, and expressions, and give examples on each of these topics.
- In Chapter 7 we define the JPL statement, and include a reference section on each of the JPL commands.
- In Chapter 8 we show how to call application code and JAM library functions from JPL procedures.
- In Chapter 9 we offer you methods to enhance the performance of your statements and procedures.



Chapter 2
Quick Start

In this chapter we use a sample JPL module to introduce you to JPL. Our goal is to help you understand the major concepts and terms before we give a more detailed technical discussion. Our sample module produces a person's full name from the first name, middle initial (if present), and last name. It is used on the screen pictured in Figure 1. This screen has four fields — `firstname`, `middleinitial`, `lastname`, and `fullname`.

```
        Name Entry Screen

First Name: _________________
Middle Initial: __
Last Name:  _________________

Full Name:
```

Figure 1: Sample screen for JPL examples.

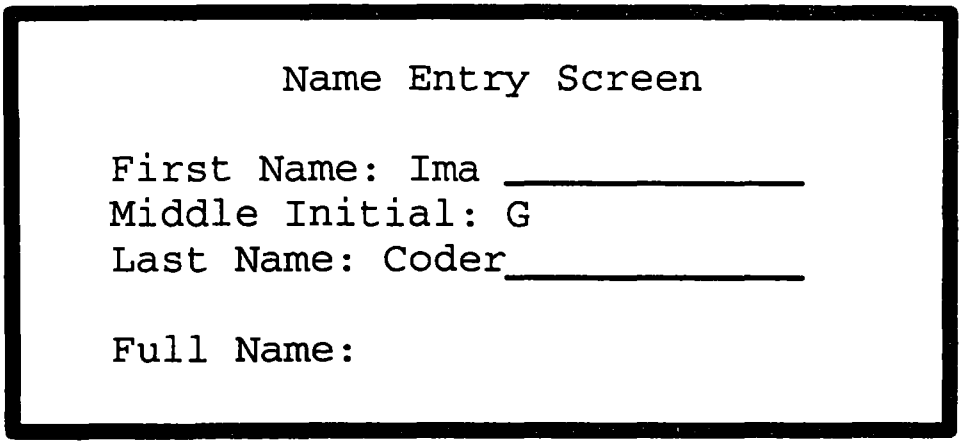


Figure 2: Sample screen for JPL examples, with data entered.

Every field has a JPL procedure window, an option under miscellaneous field edits. We entered our JPL module in the procedure window associated with the `lastname` field. JAM will execute the procedure during the field's validation process. The field JPL window is not the only place where we may enter a module. It is a sensible choice, however, for a module we would not wish to execute from other fields.

The JPL module follows:

```
if middleinitial != ""
    cat fullname firstname " " \
          middleinitial ". " lastname
else
    cat fullname firstname " " lastname
jpl special
return
#
#
proc special
# Here's a special feature.
if fullname == "George Bush"
{
    cat fullname "Mr. " fullname
    msg d_msg "Welcome Mr. President!"
}
return
```

The above JPL module consists of 17 lines. Each line in a JPL module contains only one statement. When a statement needs to be continued to one or more additional lines, a backslash (\) is used at the end of each line that is continued. The second statement, the `cat` statement, is continued to the next line. Our module has 16 statements. They form two JPL procedures. The main (first) procedure in a field module is always unnamed.

To help you better understand the example module, consider these hints:

- Each statement begins with a JPL command.
- The `if` statement tests a condition. If the condition is true, **JAM** executes the next statement. If it is false, **JAM** ignores the next statement.

  In a logical expression we use special symbols, or operators. In JPL, `!=` is the logical operator for "not equal." Therefore, the first statement says "If the middle initial field is not empty, execute the next statement."
- The `cat` (concatenate) command is a type of assignment statement. Here it concatenates five arguments and assigns the value to the field `fullname`. The arguments are three variable names (`firstname`, `middleinitial`, and `lastname`) and two quoted string constants (`" "` and `". "`).

  After **JAM** executes the first `cat` statement, the field `fullname` contains the contents of the field `firstname`, a single blank space (from the constant `" "`), the contents of `middleinitial`, a period and a single blank space (from the constant `". "`), and the contents of `lastname`.
- An `else` statement may follow an `if` statement. If the condition in the `if` statement is true, **JAM** ignores both the `else` statement and the statement following it. If the condition is false **JAM** executes the statement after the `else` statement.
- The command `jpl` calls another JPL procedure. Here it calls the procedure named `special`.
- A `proc` statement names and begins a procedure.
- A comment statement begins with the comment (#) command. We used comment statements to place blank lines between our two procedures, and to describe the procedure `special`. As of version 5.1, leading blanks are permitted on the left side of the comment symbol.
- `{` and `}` are used to group statements into blocks. In the first procedure, we executed only one statement, one of the `cat` statements, after the `if` statement. In the procedure `special`, we want to execute two statements when the condition is true. Therefore, we block those statements.
- The `msg` command displays a message on the status line.

- A `return` statement returns control to the calling procedure, or to the JAM Executive. In the procedure `special`, it returns control to the first procedure. In the first procedure, the `return` statement returns control to the JAM Executive which called this module during field validation.

Chapter 3
JPL Modules and Procedures

A module contains one or more JPL procedures. In this chapter we describe the seven different types of modules, the structure of an individual JPL procedure, and the rules for calling procedures and modules. At the end of this chapter, we provide a table which summarizes this information.

Before any statement in a module is executed, **JAM** will ensure that the module has been compiled and converted. These steps are defined as the following:

- *Compilation* A process that performs syntax checking on command words, replaces command words with tokens, and partitions a module into procedures.
- *Conversion* A process that builds internal data structures from compiled JPL module.

A module contains one or more procedures. Procedures may be named or unnamed.

- *named* JPL procedures are named using the JPL command word `proc`. A module may contain any number of named procedures. A named procedure may always be called by itself or by other procedures in the same module. Some named procedures may also be called from outside their modules.
- *unnamed* Only one unnamed procedure is permitted in a module. If a procedure is unnamed, it must be the first procedure in the module. Some unnamed functions are called and executed automatically by **JAM**. Others are executed when the module name is called.

The type of module you use determines the scope of its procedures.

- *entry point* If a module has an entry point, it contains at least one procedure which may be called explicitly from outside the module. Unnamed and named procedures are entry points to modules. An unnamed procedure is called by its module's name, and a named procedure is called by its procedure name.

These definitions are important to your understanding of the material in this chapter.

3.1 JPL MODULES

We discuss each of the module types in the following sections. If you are learning JPL for the first time, you should read the sections on field, screen, and file modules, and skip the remaining module types for now. When you are more familiar with JPL, read the sections on public, load, library, and memory–resident modules.

3.1.1 Field Modules

Every field on a screen has a JPL procedure window, an option under miscellaneous field edits. JAM saves and compiles a JPL module entered here when you XMIT from the JPL procedures window. JAM converts the module when it is called during the field's validation.

The first procedure in a field module must be unnamed. JAM automatically executes the first procedure in this module during field validation, after executing any validation function. You may also have named procedures in this module, but these procedures can be executed only if called by another procedure in the module.

There are no entry points to this module. Therefore, you can never directly call any field module's procedures from outside the module.

Since the module is stored with the field, copying the field to (or from) a clipboard or the data dictionary also copies the JPL module.

This module is useful for a function designed for a particular field, and for a function that you want executed whenever the user tabs out of the field.

3.1.2 Screen Modules

A JAM screen has a JPL procedures window, an option in the screen edits window. JAM compiles and saves all the procedures in this module when you XMIT from the JPL procedure window. The module is saved with the binary screen file.

When the screen is opened, JAM converts the module, and executes the first unnamed procedure, if any. This procedure receives as parameters the name of the screen and the K_ENTRY bit. The procedure is executed only when the screen is first displayed. The first unnamed procedure is *not* executed when the screen is exposed by virtue of another window being closed.

While the screen is active (i.e., displayed on top), every named procedure in this module is a possible entry point.

The screen module is useful if several fields on the same screen use the same function. In addition, every screen entry and exit function, field, group, and control string function, needed for this screen may be stored and called from this window.

You can test screen module procedures in application mode but not in test mode.

3.1.3
File Modules

You may create a JPL file module with any text editor. This module is created and stored outside an application's binary screen files.

You may also create a file module with the "file access" option in a JPL procedure window. Choosing "write" copies the contents of the window to a file which you name.

An ASCII file module is compiled each time it is called. Runtime compilation may be eliminated by using the **JAM** utility `jpl2bin` to compile a file module and save it in a binary file. If memory is tight in your application, you may wish to compile your file modules with `jpl2bin`, and then stub out the runtime JPL compiler. Both ASCII and binary file modules are converted when called.

A file JPL module has one entry point — its unnamed first procedure. You must use the name of the file to call this procedure. A file module is accessible if it is in the current directory or in a directory specified by the library function `sm_initcrt` or by `SMPATH`. Scope of JPL modules is discussed fully in section 3.4 on page 14, Calling JPL Procedure and Modules.

The wider scope of file modules makes them a useful alternative to screen and field modules, but there are some disadvantages. Modules stored in files are processed more slowly because **JAM** does not automatically compile the module when you exit and save the file. Unless you use the utility `jpl2bin`, **JAM** must compile the module every time it is called. Since file modules are stored outside the **JAM** executable, they are also more difficult to protect from loss or accidental editing by the end–user. The next four modules are possible solutions to these problems.

3.1.4
Public Modules

JPL has two commands which affect how a file module is used. `public` is one of these JPL commands. When a `public` statement is used in a module, it has one or more

module names as arguments. When it is executed at runtime, it performs the following steps: 1) reads the named JPL module, 2) compiles it, if necessary, 3) converts it, and 4) executes the first procedure if it is unnamed. While the module is public, every named procedure in the public module is a possible entry point to the module.

A public module combines the features of a screen module and a file module. As in a screen module, every named procedure is a possible entry point to the module, but like a file module, a public module is not limited to any one screen. Instead, the procedures in a public module are available throughout the application.

A public module is removed from memory by using the JPL command `unload`.

You may name any ASCII or binary JPL file module as an argument in a `public` statement. Further below we discuss two other module types — library and memory–resident modules. These may also be made public.

3.1.5
Load Modules

`load` is another JPL command which affects how a module is used. It has one or more module names as arguments. When a `load` statement is executed at runtime, it performs these three steps: 1) reads a JPL module, 2) compiles it if necessary, and 3) converts it. A load module remains in memory until you release the memory with an `unload` statement. Until then, every call to the module executes it without any additional compilation or conversion.

A load module has one entry point — its first unnamed procedure. You must use the name of the module to call this procedure.

You may create load modules from any ASCII or binary JPL file module, or a library module (see below). A load module has the same name as its source module (the file or library module named in the `load` statement).

3.1.6
Library Modules

JAM provides a utility, `formlib`, for creating application libraries. In addition to screens, you can also store JPL modules in a library. Use a text editor to create the file module, compile it with the JAM utility `jpl2bin`, and add it to the library with the utility `formlib`. (See section 9.1.2 on page 98, and also the *Utilities Guide*, for a detailed explanation.) Conversion occurs each time you call the module.

A library must be opened before you can use any of the screens or modules stored in it. Libraries are usually opened in an application's `jmain` module with the library function `sm_l_open`.

A library module has one entry point — its first unnamed procedure. You must use the name of the library module to call this procedure.

Using libraries reduces both I/O time, and the number of files required for distribution. Libraries are inconvenient if you need to edit procedures, since you must edit the module, recompile it `jpl2bin`, and reinstall the module to the library with `formlib`.

See the **JAM** *Programmer's Guide* for more information on using libraries.

3.1.7
Memory–Resident Modules

You can install JPL modules in an application's memory–resident list. Use a text editor to create the file, compile the module with the utility `jpl2bin`, convert the binary file to a C language character array with `bin2c`, and then install it with the library function `sm_formlist`. You must recompile your application after creating or editing a memory–resident list. See section 9.1.3 on page 98, and the also **JAM** *Programmer's Guide* for more information.

Since the module must be compiled with `jpl2bin` before you install it in the memory–resident list, no compilation is needed when you call the module. Conversion occurs, however, each time you call the module.

A memory–resident module has one entry point — its first unnamed procedure. You must use the name of the memory– resident module to call this procedure.

Making a JPL module memory–resident reduces I/O time and makes it a part of the JAM executable. The module is held in memory during the life of the application.

Once you make a JPL module memory–resident, it is more difficult to edit. If you make changes to a memory–resident module, you must edit an ASCII version of the module, recompile the module with the utilities `jpl2bin` and `bin2c`, and then recompile the application program.

3.2
CREATING MODULES

Modules are created using the Screen Editor or a text editor. We discuss the characteristics of these editors below.

3.2.1

JPL Procedures Windows

Field and screen JPL modules are edited with `jxform`, inside JPL procedure windows. A JPL procedure window is an editing screen with some special features which are available on every JPL procedure window.

A JPL procedure window is actually a scrollable and shiftable array, permitting 500 lines of code, each up to 125 characters long. Text is entered directly from the keyboard. Lines may be inserted with the logical key INSL, and deleted with DELL. (On a PC, for example, we often map these keys to Alt–I and Alt–D.) Use the modkey utility if you do not know the key mapping on your terminal. The JPL procedure windows also feature a "file access" option. When you open a JPL procedure window for editing, pressing PF2 displays a file access window, where you may select the read or write option and where you may enter a file name. The "read" option reads text from a file and inserts it in the JPL procedure window at the line before the cursor. The read option does not copy tabs to the JPL window; instead, it replaces them with spaces. The "write" option writes the text in the current JPL window to a file. In the text file, it terminates each line from the JPL window with a new–line character.

You may enter any number of named procedures in a procedure window.

Pressing XMIT inside a JPL procedures window saves and compiles the JPL module, and closes the JPL procedure window. If a command word is misspelled, **JAM** will stop compilation and display an error message. Every statement must begin with a valid JPL command word, or **JAM** will not save the module. Messages for other errors, such as undefined variables or division by zero, are displayed at run–time.

3.2.2

JPL Text Files

The other JPL modules are created with a text editor. **JAM** will display error messages for misspelled commands words when it compiles the module. An ASCII file module is compiled at runtime, a binary file when `jpl2bin` is used, a load module when `load` is executed, and a public module when `public` is executed. Except for load and public modules, conversion of JPL text files occurs at runtime.

Text files should be named according to the rules of your operating system. **JAM** does not append any extension to ASCII JPL files. The default extension for binary JPL files is `bin`.

3.3 JPL PROCEDURES

3.3.1 Structure of Procedures

A JPL procedure has one or more JPL statements. Unless a procedure is the first one in the module, it must be named. A named procedure begins with a `proc` statement.

`vars` and `parms` statements define variables in JPL procedures. Those in a named procedure are local to the procedure, while those in an unnamed procedure are global to all procedures in the module. Consider the following example.

```
vars global
cat global '0'
vars another_global
cat another_global '1'
vars this_is_also_global

proc first_screen_proc
vars local
    ...
```

A procedure ends at the bottom of a module, or at the statement before the next `proc` statement, whichever comes first. A procedure returns to its caller when its end is reached. The following example JPL module contains two procedures, one unnamed and the other named `warning`:

```
# This procedure is unnamed.
if cost > 100
    jpl warning
# The next procedure is named.
proc warning
msg emsg "The cost is very great."
```

Statement execution within a procedure begins with the first statement of the procedure, and continues sequentially until the end of the procedure is reached, or until a `return` statement is executed. If a JPL procedure encounters an error during execution, it aborts and returns –1. The order of statement execution may be altered by the `if`, `for`, `while`, `else`, `break` or `next` statements. The first four of these statements may be followed by the blocking statements, `{` and `}`, to conditionally execute a block of statements. For example:

```
if cost > 1000
{
    math exceptions = exceptions + 1
    msg emsg "The cost is very great."
}
```

3.4

CALLING JPL PROCEDURES AND MODULES

Any module may begin with a unnamed procedure. JAM will automatically execute this unnamed procedure:

- in a field module when the field is validated.
- in a screen module when the screen is opened.
- in a module when it is put in memory by a `public` statement.

A procedure in a module may call any named procedure in the same module. A procedure is called by its procedure name, using a `jpl` statement.

The following procedures are entry points to a module and may be executed by an explicit call (`jpl`, `sm_jplcall`, `^jpl`, etc.):

- a named procedure in the screen module of the active screen (the screen currently displayed); call it by its procedure name.
- a named procedure in any public module; call it by its procedure name.
- the first procedure (which must be unnamed) in any load module, memory–resident module, library module, or file module; call it by its module name.

When you call a JPL function, there is no way to indicate whether you are using a procedure name or a module name. Therefore, JAM looks for the named routine by first examining the names of all available procedures, and then the names of all available modules. Below is a list of the order in which JAM examines the names of procedures and modules. If JAM finds the routine it stops the search and executes the routine. If it is unable to find the routine it will display an error message.

1. a named procedure in the same module (if a module is being executed).
2. a named procedure in the screen module of the screen now displayed.
3. a named procedure in a public module.
4. a load module.
5. a memory–resident module.
6. a library module in an open library.

7. a file module in the current directory.
8. a file module in a directory specified by the library function `initcrt ()`.
9. a file module in a directory specified by `SMPATH`.

For example, you might call a JPL routine with the statement

```
jpl totals
```

Once JAM finds a routine named `totals`, the search stops. If `totals` is a procedure, JPL will execute the procedure. If `totals` is a module, JPL will execute the first procedure in the module, if the procedure is unnamed. If JAM is unable to find any procedure or module named "`totals`" it will display an error message.

Note that a load module has scope which is higher than any module which may be loaded. Since a load module has the same name as the module named in the `load` statement, JAM will never execute a memory–resident, library, or file module when a load module with the same name is currently in memory. If you `unload` the module, and then call it, JAM will not find the module in memory. Therefore, it will search the memory–resident list, the open libraries, and the default directories until it finds the module. Similarly, a memory–resident or library module always has the same name, but a higher scope, than the file module used to create the library or memory–resident entry.

When a `public` statement is executed, JPL puts a copy of the module in memory but permits only procedure names as entry points to the public module. If `total_charges` is the following file module,

```
vars total

proc tax_total
parms amount state
vars tax
if state...
....
math total = amount * tax

proc ship_handling
parms amount items weight
vars charge
...
math total = amount + charge

proc rush_fee
...
math total = amount + charge
```

and you use these two statements,

```
public total_charges
jpl total_charges
```

JAM will execute the statement `vars total` twice.

When JAM executes the `public` statement, it compiles, converts, and puts the module in memory, and then executes the module's first, unnamed procedure. When it tries to execute the `jpl` statement, however, it will not find a procedure named `total_charges` in memory. (The procedure names in memory are `tax_total`, `ship_handling`, and `rush_fee`.) Therefore, it will continue searching until it finds the file, which it compiles and converts, and then it executes the first, unnamed procedure. In short, since the module name is not an entry point to a public module, the public module will not prevent you from executing the copy on disk or in a library or in the memory–resident list. This is significant if you are using variables global to a module. In this example, `total` is global to all the procedures in its module, but a change made to `total` in the public module does not affect `total` in the file module.

Keep in mind this scoping order when you are naming procedures and modules. In particular, if you are using different types of modules, check that screen and public procedure names do not conflict with the names of any file, library, or memory–resident modules that you wish to call.

3.5
SUMMARY OF JPL MODULES

Module Type	*Module Location*	*Entrypoints*	*Compilation*	*Conversion*
field *(first procedure must be unnamed)*	JPL procedure window associated with a field. Stored in the screen binary.	None; first procedure called by JAM during field validation.	When saved (by pressing XMIT in JPL window).	When called by JAM during field validation.
screen *(screen modules cannnot be tested in the screen editor)*	JPL procedure window associated with a screen. Stored in the screen binary.	All named procedures while screen is active; use proc names. (If first procedure is unnamed, JAM calls it when screen is opened.)	When saved (by pressing XMIT from JPL window).	When screen is opened.

Module Type	*Module Location*	*Entrypoints*	*Compilation*	*Conversion*
file	File. A file is created with a text editor. It may be compiled and saved to a binary file with the utility `jpl2bin`.	First, unnamed procedure; use file name.	With `jpl2bin`, or when called.	When called.
memory–resident	Member of a memory– resident form list. Stored with the JAM executable.	First, unnamed procedure; use name of memory list entry.	Must be compiled with `jpl2bin` before it is added to the memory– resident list.	When called.
library	Member of a library. Stored in a library.	First, unnamed procedure; use name of library entry.	Must be compiled with `jpl2bin` before it is added to the library.	When called.
load	File, memory–resident member, or library member. The file or member is named in a `load` statement. It is held in memory until unloaded.	First, unnamed procedure; use name of load module.	With `jpl2bin`, or when the `load` statement is executed.	When `load` is executed.

Module Type	*Module Location*	*Entrypoints*	*Compilation*	*Conversion*
public	File, memory–resident member, or library member. The file or member is named in a `public` statement. It is held in memory until unloaded.	All named procedures; use `proc` names. (If first procedure is unnamed **JAM** calls it when `public` is executed.)	With `jpl2bin` or when the `public` statement is executed.	When `public` is executed.

3.5.1

Calling JPL Routines from a Control String

You may call a JPL procedure or module with a control string. In the control string window of the Screen Editor, enter a caret, the command `jpl`, the name of the JPL procedure or module, and any arguments. You may also use a target list.

```
^jpl function-name {arg}...
^(target list)jpl function-name {arg}...
```

Do not put blank spaces after the caret.

JAM expects a control string function to return an integer. **JAM** compares the returned value against the target list. If it does not find a match, it ignores the value. See the *Author's Guide* for directions on creating a target list.

Since control strings are used to define flow control within an application, you might write a JPL procedure that evaluates the actions of the end–user, and returns an integer that will determine the next form or window to be displayed.

Remember that control strings are not executed in test mode. Use application mode to test any JPL procedures called by a control string.

3.5.2

Calling JPL Routines from a JPL Module

JPL procedures may call other JPL procedures or modules, according to the scoping rules defined on page 14. The syntax is

```
jpl function-name {arg} ...
```

In the list of arguments, you may pass the values of variables local to the calling procedure to variables defined in ***function–name***. You may also pass field or LDB entries as arguments. A `parms` statement will receive the values of the arguments being passed.

```
proc function–name
parms arg ...
```

JAM will update LDB entries or current screen occurrences that are given new values inside a procedure. To get the value of an integer returned by a procedure, use a `retvar` statement in the calling procedure.

```
vars v
retvar v
jpl test :name
#         v is either 0 or 1, according to the
#         value returned by procedure "test"
if v
#  normal_process
{
}
else
#  process_Johnson
{
}
return

proc test
parms n
if n == 'Johnson'
     return 0
else
     return 1
```

The `retvar` variable may also be a field or LDB entry.

When you are using the same JPL procedure more than once, it is convenient to write one procedure, and pass the values when you call the procedure. This way, a procedure does not have to be "hard–coded" for every field or screen that uses the procedure.

3.5.3
Calling JPL Routines from a Field Function

JAM provides the ability to attach four functions to every field — an entry function, a validation function, a JPL field module, and an exit function. Any or all of these functions may invoke JPL functions.

To use a JPL function as a field entry, validation, or exit function, enter the command `jpl` and the function name in the miscellaneous, field functions window.

```
jpl function-name
```

JAM automatically passes four arguments to the procedure you are calling. These arguments are,

field-number, field-contents, occurrence-number, flags.

A `parms` statement is required in the called procedure if it is to access the parameters.

For example, in the attached function window, you might call a JPL function to perform a field exit function. If the call is,

```
jpl fld_xt
```

the procedure `fld_xt` might begin,

```
proc fld_xt
parms num val occ flg
if val = 'MR'
     cat sex 'M'
else
     cat sex 'F'
return
```

The parameter `num` receives the number of the field calling the exit function. `val` receives the contents of the field. `occ` receives the current occurrence number, which is 1 if the field is not an array. The parameter `flg` receives bit values. Unless you are using the same function for field entry and exit, you will probably ignore the flag argument. You do not need to define trailing parameters you will not use. For example, you may use this `parms` statement:

```
parms number contents occurrence
```

While you do not explicitly call a field JPL module by name (it is unnamed), JAM executes the module during field validation processing. The module receives the same arguments from the field as do the other field functions. A `parms` statement in the module should establish parameters like those used in the examples above.

The JAM *Programmer's Guide* contains a detailed explanation of field arguments. If you choose to use the same JPL procedure as a field entry and exit function, you will need to use the ***flags*** argument. Since this is a bit value, see section 6.3.3 on page 42 about JPL bitwise operators, or see the *Programmer's Guide*.

3.5.4

Calling JPL Routines from a Group Function

A group entry or exit function may call JPL procedures, with the verb `jpl`. It passes two arguments by default,

group–name flag.

You might, for example, write a JPL procedure to turn on and off the menu toggle upon entering and exiting a field group.

As in field arguments, ***flag*** is a bit value. You will need to use bit operators (see section 6.3.3 on page 42) to use this argument.

3.5.5
Calling JPL Routines from a Screen Function

Screen entry and exit functions may call JPL procedures as well. A JPL screen entry function might, for example, concatenate LDB elements to place a title on a screen.

The calling syntax is

`jpl` ***function–name***

It passes the arguments,

screen–name flag.

If parameters are declared in the procedure with a `parms` statement, the first parameter variable receives ***screen–name***, if it is available. The names of memory–resident screens, for example, are not available; a null string is passed instead. The calling statement ignores any values returned from the procedure.

3.5.6
Calling JPL Routines from Application Code

JPL procedures may also be called from application code using a library function. In C for example, the library function

`sm_jplcall ("`***function–name*** `{`***arg***`}..."`)

executes a JPL procedure.

The reference section of the JAM *Programmer's Guide* explains all of the JAM library functions for your application language.

Chapter 4
JPL Variables

Variables hold values. This chapter discusses how JPL variables are defined and referenced. This chapter will also help you better understand Chapter 5, "The Colon Preprocessor," and Chapter 6, "Data Types, Operators, and Expressions."

4.1
DEFINITION OF JPL VARIABLES

There are three kinds of variables available to you when you are using JPL. The first is screen variables — fields and groups. The second is local data block (LDB) entries. The third is JPL variables defined with `vars` and `parms` statements inside a module. The names of JPL variables are constructed with the same rules that apply to field and LDB names. We recommend using any combination of 1 to 31 letters, digits, and underscores, that does not start with a digit. The rules also permit the use of two special characters, the dollar sign ($) and period (.), and you may use these when necessary.

When a variable is "defined" storage is allocated. A JPL variable must be defined by a `vars` or `parms` statement before the variable is used. In some instances, you may also need to "declare" how a variable will be used. For example, after defining a variable you may declare the variable as a return variable with a `retvar` statement. The definition statement tells JPL how much space to allocate for the variable while the declaration tells JPL what to store in the variable (i.e., the integer value returned by a function). Below we discuss the two commands. `vars` is a definition command, and `parms` is both a definition and declaration command.

A `vars` statement defines one or more JPL variables. The value of a newly defined variable is the null string (""). In a `vars` statement you may also define a variable's number of occurrences (array size) and/or its occurrence size (number of characters).

The default number of occurrences is the minimum (one occurrence). The default occurrence size is the maximum variable size, 255 bytes. JPL allocates space for the size you specify, plus an additional byte for the terminating null.

When you use a `jpl` statement to call a procedure, you may need to pass values to the procedure. A parameter in the procedure receives the value passed from a `jpl` statement. A `parms` statement will define a variable if it does not already exist, declare the variable as a parameter, and assign it the value of an associated argument. If the variable is an array, however, then you must define its number of occurrences and its size before declaring it in a `parms` statement.

Some example `vars` and `parms` definitions follow. The number of occurrences is in square brackets. The occurrence size is in parentheses.

```
vars acctno
vars fullname[100](30)
vars age[100](3)
vars qty cost extension
vars list[25]
parms querytype(3)
```

In the next example, the `vars` statement defines `city` as an array with three occurrences. The `parms` statement declares the three occurrences of `city` as three parameters. Since `x` has not been defined, it defines `x` as a single occurrence with size 2, and then declares `x` as a parameter.

```
vars city[3]
parms city[1] city[2] city[3] x(2)
```

In all cases, a statement cannot reference a JPL variable until JPL executes the variable's definition statement, either `parms` or `vars`. JPL variables may be defined as either global to the procedures in a JPL module, or local to one JPL procedure. `vars` or `parms` statements before a module's first `proc` statement define variables global to all the procedures in the module. Definition statements in a named procedure define variables local to the procedure.

4.2
SCOPE AND LIFETIME OF VARIABLES

In this section we discuss a variable's scope and lifetime. Scope describes the part of the application over which the variable is defined. Therefore, scope describes when and where you may reference the variable. Lifetime describes the length of time that the variable exists. When you assign a value to a variable it keeps this value until you assign it another, or until its lifetime ends. When its lifetime ends, JAM re–allocates the variable's memory.

The scope of a variable defined within a named JPL procedure is local to that procedure. That is, it may only be referenced by JPL statements that are part of that procedure. The lifetime of such a variable is the time in which that procedure is in a state of execution.

The scope of a variable defined in the unnamed procedure at the top of a module is global to the JPL module. It may be referenced by any JPL statements that are part of the module, but not by JPL statements in other modules. The lifetime of such a variable is the same as the lifetime of the JPL module.

The scope and lifetime of variables defined in JPL is summarized below:

Definition Location Within JPL Module	*Scope*	*Lifetime*
within JPL procedure	JPL procedure	execution of JPL procedure.
within a module, but prior to any JPL `proc` statement	JPL module	field, file, load, memory–resident, library modules: execution of module.
		screen module: until screen is closed.
		public module: until module is unloaded.

Global module variables in field, file, load, memory–resident, and library modules are defined each time the module is executed. After JPL executes the module and returns control to the **JAM** Executive, the memory for the global variables is re–allocated.

Global module variables in a screen module are defined when the screen is opened. **JAM** re–allocates the memory of these variables when the screen is closed.

Global module variables in a public module are defined when the `public` statement is executed. The memory for these variables is re–allocated when the module is unloaded (named in a `unload` statement).

Field and group variables are local to the screen. JPL statements may reference any field or group name while the screen is active (displayed on top). The lifetime of these variables is the length of time the screen is on the window stack.

LDB entries are accessible throughout the lifetime of the application.

4.3
REFERENCING VARIABLES BY NAME

Different variables may share the same name. When you reference a variable by name, such as in `cat fullname "John Doe"`, you need to know which instance of `fullname` is being referenced. It could be a local JPL variable defined within the procedure, it could be a global JPL variable in the module, it could be a screen field or group, or it could be a LDB entry.

The JPL processor determines the referenced variable by searching the following places, in order, until the named variable is found:

1. the procedure that contains the reference. Note that the variable will be found here only if the defining `vars` or `parms` statement has already been executed.
2. the module that contains the reference.
3. the fields or groups on the screen.
4. the local data block entries.

References to variables may include a specification of an occurrence number in brackets, following the variable name itself. Spaces are permitted between the variable name and the left bracket. For example, if `customer` is an array with five occurrences, then `customer [3]`, refers to the third occurrence of the array.

4.4
REFERENCING VARIABLES BY FIELD NUMBER

References by field number are specifically treated as references to fields on the screen. For example, `#5` refers to the fifth field on the active screen. From field functions written in JPL you may also make a reference relative to the current field (the field on which the cursor is positioned). For example,

`#+1`	refers to the field following the current field.
`#-1`	refers to the previous field.
`#+0`	refers to the current field.

Be careful when using field numbers, since field numbers may change when the screen is edited. In general, relative references by field number are safer than absolute refer-

ences by field number. In particular, `#+0` (or `#-0`) is always a safe and efficient method of referencing the current field.

Relative field numbers are supported only for field functions. They do not work in screen or control string functions written in JPL. Prototype the library function `sm_getcurno` if you need to know the cursor position in a JPL screen or control string function.

4.5
REFERENCING GROUP VARIABLES

The two kinds of groups in JAM are radio buttons and checklists. One occurrence may be selected from a radio button, while zero or more may be selected from a checklist. Each group has a name, which may be referenced to determine an end–user's selections.

A radio button name has one occurrence, and a checklist name has an occurrence for each occurrence in the checklist. The radio button name is set to the group occurrence number of the user's selection. In the example below, `day` is a radio button with 7 occurrences.

```
MONDAY   TUESDAY   WEDNESDAY   THURSDAY   FRIDAY   SATURDAY   SUNDAY
```

If the end–user selects `THURSDAY`, then `day` equals 4. While the screen is active, you may use `day` as a variable in a JPL procedure. If all the occurrences in the radio button belong to an array named `day_array`, then the variable `day_array [day]` equals `THURSDAY`.

The first occurrence of a checklist variable is set to the group occurrence number of a selection, or set to null if none are selected. The second occurrence is set to the number of another selection, or null if the end– user did not make a second selection, and so on. The group occurrence numbers of the selections are stored sequentially (although not necessarily in any particular order) in occurrences of the checklist variable. Consider the following example. `state` is the name of an array with three occurrences.

```
MAINE   VERMONT   NEW HAMPSHIRE
```

`s_list` is a group with three occurrences — the three occurrences of `state`. Since `s_list` is a checklist, the end–user may choose zero or more selections. If he selects `VERMONT` and `NEW HAMPSHIRE`, then `s_list[1] = 2`, `s_list[2] = 3`, and `s_list[3] = 0`.

You can use a loop to count the number selections or display the selections. For example,

```
vars maxoccur count i
cat maxoccur '3'
cat count '0'
for i =1 while i <= maxoccur step 1
{
   if s_list[i] == ''
      break
   else
      math count = count + 1
}
msg emsg "User made :count selections."
```

Chapter 5
The Colon Preprocessor

Colon expansion provides great programming flexibility, since the JPL programmer is not restricted to using variables only where variables are normally permitted in JPL statements. We use the term "preprocessor" to emphasize that colon expansion is an independent process which JAM performs before executing each JPL statement. After a module has been compiled and converted, but before a statement is executed, the colon preprocessor scans the statement and replaces each colon variable with its current value. A colon variable name begins with a colon and ends with a blank or another character that cannot be expanded. After making the substitution, the colon preprocessor then hands the statement over to the JPL statement processor for execution.

For example, suppose that the value of `acctno` is `91956`. The execution of the statement

```
msg emsg "I cannot find :acctno."
```

would result in the display of the message "`I cannot find 91956.`", even though the syntax of the `msg` statement does not permit variables inside of the message string itself. In contrast,

```
msg emsg "I cannot find acctno."
```

would result in the display of the message "`I cannot find acctno.`"

You may also use colon expansion where variables or procedure or module names are permitted. In fact, any statement that permits a variable as an argument also permits a colon–expanded variable. For example,

```
cat fruit 'a'
cat a 'apple'
msg emsg :fruit
```

would display `apple` (rather than `a`) on the status line.

5.1 REFERENCING VARIABLES FOR COLON EXPANSION

A variable will be recognized by the colon preprocessor only if it is immediately preceded by a colon (:). To avoid having a colon expanded, you may either precede the colon with another colon (::), precede the colon with a backslash (\:), or follow the colon with a space (:).

In the first two cases, the colon preprocessor discards the first colon and the backslash. In the third case, the colon and following space are preserved. Once a variable is recognized by the colon preprocessor, the rules in Chapter 4 for determining the proper reference of the variable (e.g. local procedure variable, global module variable, screen field or group, LDB entry) apply. The Chapter 4 rules for determining the value of the variable also apply, except in the case of an array reference with no specified occurrence number. In that case, the colon expansion concatenates all the non–blank array occurrences, separating each pair of occurrences with a blank space (see example below).

Determining the referenced occurrence of a colon–expanded variable can be tricky. Consider the following example:

```
vars xyz[3] alpha[3]
vars v w
vars x1 x2 x3 x4 x5

cat alpha[1] "bits"
cat alpha[2] "centuri"
cat alpha[3] "rays"
cat xyz[1] "alpha"
cat xyz[2] "beta"
cat xyz[3] "gamma"
cat v "alpha"
cat w "xyz"

cat x1 xyz[3]
#    Now x1 = gamma
cat x2 :xyz[1][3]
#    Now x2 = alpha[3] = rays
cat x3 :v [3]
#    Now x3 = alpha [3] = rays
cat x4 ":xyz"
#    Now x4 = ":xyz[1] :xyz[2] :xyz[3]" =
#               "alpha beta gamma"
cat x5 :v[3]
#    Now a JPL error occurs because v[3] does not exist.
```

Normal variable substitution replaces `xyz[3]` with the value of the third occurrence of `xyz`. When the `cat` is executed, the value of `x1` is `gamma`.

The first substitution by the colon preprocessor occurs in the statement `cat x2 :xyz[1][3]`. The colon preprocessor replaces `:xyz[1][3]` with `alpha[3]`. When the `cat` is executed, the value of `x2` becomes `rays`.

In the next statement the colon preprocessor replaces `v` with `alpha`. `x3` then equals the third occurrence of `alpha`, which is `rays`. The space between the variable name `v` and the index brackets is significant. The space prevents the colon preprocessor from trying to expand the third occurrence of `v`.

In the next statement the colon preprocessor concatenates all the non–blank occurrences of xyz, separating the occurrences with single blank spaces. If `:xyz` was not enclosed in quotes, JAM would display an error message because `beta` and `gamma` are not variables. In the last statement the colon preprocessor tries to replace `:v[3]` with the third occurrence of `v`. Since `v` has only one occurrence, JAM displays an error message.

5.1.1
References with Substring Specifiers

Substring specifiers may also be used with colon expansion. If there is a substring specifier immediately after a variable name, the colon preprocessor will extract the specified characters from the value of the variable. If there is a blank between the variable name and the specifier, the colon preprocessor ignores the specifier, and JPL interprets the specifier when it executes the statement.

```
vars abc xyz xy m1 m2
cat xy "New Zealand"
cat xyz "Belgium"
cat abc "xyz"
cat m1 :abc(1,2)
cat m2 :abc (1,2)
```

When the procedure is executed, the value of `m1` is "`New Zealand`," and the value of `m2` is "`Be`."

When the colon preprocessor examines

```
cat m1 :abc(1,2)
```

it replaces `:abc(1,2)` with `xy` and returns control to the JPL statement processor. The JPL statement processor replaces `xy` with its value, "`New Zealand`," and assigns the value to `m1`. (If `xy` were not a variable, JAM would display an error message).

When the colon preprocessor examines

```
cat m2 :abc (1,2)
```

it replaces `:abc` with `xyz`. Since there is a blank space after the variable, it returns control to the JPL statement processor. The JPL statement processor replaces `xyz` with its value, "`Belgium`," extracts the substring `Be`, and assigns "`Be`" to `m2`.

5.1.2

Colon–Expanded Arguments in Invocation Statements

Some JPL commands do not recognize variable arguments. They are the invocation commands (`atch, call, jpl, system`), the **JAM/DB*i*** commands (`dbms, sql`), and the commands `load` and `public`. They will use variable names as string constants unless you colon–expand the variables. For example,

```
proc money
vars t1 t2
cat t1 "nickels"
cat t2 "dimes"
jpl display t1 t2

proc display
parms p1 p2
msg emsg p1 " " p2
```

The `jpl` statement would pass `t1` and `t2` as string constants to `display`. The `msg` statement would put

```
t1 t2
```

on the status line, because variable substitution is performed in `msg` statements, but not in invocation statements. (Colon expanding p1 and p2 would cause an error message because `t1` and `t2` are local to `money`.) To pass the contents of `t1` and `t2`, use this statement

```
jpl display :t1 :t2
```

Then the `msg` statement would put

```
nickels dimes
```

on the status line.

If the value of colon expanded variable is a string with one or more embedded spaces, the spaces in the string will be treated as delimiters between the arguments. For example,

```
vars str
cat str "a b         c"
jpl display :str

proc display
parms x y z
msg emsg x y z
```

Colon preprocessing on the `jpl` statement produces,

```
jpl display a b      c
```

When the `jpl` statement is executed it passes three arguments, `a`, `b` and `c`, to the function `display`. The `msg` statement would put

```
abc
```

on the status line.

To pass the contents of `str` as a single argument, use these statements:

```
jpl display ":str"

proc display
parms x
msg emsg x
```

Then the `msg` statement would put

```
a b       c
```

on the status line.

5.1.3
References in Parentheses

Parentheses may be used when referencing variables for colon expansion. JAM provides this feature to make writing correct references easier, and to permit proper expansion of variables that are followed by characters. The parentheses are placed around the variable name (or field number). In this example,

```
vars ref alpha[3]
cat alpha[1] "bits"
cat alpha[2] "centuri"
cat alpha[3] "rays"
cat ref "alpha"
cat x4 :(ref)[3]
#     Now x4 = rays
```

the colon preprocessor replaces `:(ref)` with `alpha`, and JPL replaces `alpha[3]` with `rays` and assigns this string to `x4`.

5.2
FORCING RE–EXPANSION

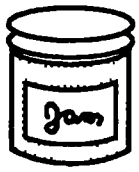
Normally, the colon–expanded text is not revisited by the colon preprocessor, even if it contains another colon– reference to a JPL variable. For example, the following code results in the display of the message "Thank George it's :day"

```
vars day
cat day "Friday"
vars period
cat period "\:day"
msg emsg "Thank George it's :period"
```

To display the message "Thank George it's Friday", you could use the colon re–expansion operator, * (asterisk). You'd get your wish if the msg statement was replaced with:

```
msg emsg "Thank George it's :*period"
```

The colon expansion preprocessor actually works from right to left. Normally, it doesn't try to re–expand expanded text. When the re–expansion operator is encountered, the colon preprocessor re–starts the expansion at the right–most character of the expanded text. Normal processing resumes at that time. Therefore, it is permissible to nest the use of colon re–expansion (although it is difficult to imagine needing this!).



Chapter 6
Data Types, Operators, and Expressions

Data types describe how JPL uses the values of variables and constants. Operators specify what is done, or how the variables and constants are manipulated. Expressions combine variables and constants to produce new values. An understanding of these topics will provide the foundations for writing JPL procedures.

6.1
DATA TYPES

The data type of a variable or expression in JPL is not determined by declaration. Instead, type is determined by context, that is, by value and usage. JPL stores all values as character strings, and performs conversions when necessary. There are four data types:

- *string* A string is zero or more characters. A string requires no conversion. (There is no defined limit on string length in JPL.)
- *integer* A sequence of digits that does not contain a decimal point. It may begin with a plus or minus sign. Conversion is to integer.
- *numeric* A numeric begins with a digit, a plus or minus sign, or a decimal point. Numerics are converted to floating point.
- *logical* A string, integer, or numeric may be evaluated as a logical, that is, as true or false. A string is a logical true if it begins with the value of message entry `SM_YES` (often "`y`" or "`Y`"). It is false if it begins with any other character. A numeric or integer is a logical false if it is zero, and a logical true for all other numbers.

6.2 CONSTANTS

There are several kinds of constants in JPL. We describe them in the subsections below.

6.2.1 Integer Constants

A integer constant consists of an optional plus or minus sign followed by a sequence of digits. JPL supports only decimal integers. Binary, octal or hexadecimal integers are not available (except as arguments to prototyped functions, see page 94).

6.2.2 Numeric Constants

A numeric constant consists of an optional plus or minus, followed by an integer part, a decimal point, and a fraction part. The integer and fraction parts are sequences of digits.

Since JPL performs data type conversions when necessary, you may represent a numeric constant with only the integer part.

6.2.3 Date Constants

A date constant is a literal date enclosed in parentheses. To be recognized, it must use the date format specified in the message file entry `SM_CALC_DATE`. The default in the message file is `%m/%d/%4y`, which is `MON/DATE/YR4` in mnemonics. The `@date` operator converts a date constant to a numeric by counting the number of days between the date constant and January 1, 1753 (the standard for date calculations).

6.2.4 String Constants

A string constant in most programming languages is a sequence of zero or more characters surrounded by quote symbols. JPL, however, distinguishes between two types of string constants, quoted and unquoted.

Quoted string constants

A quoted constant contains zero or more characters. At runtime, JPL strips off the quote characters.

Both single and double quote symbols are permitted, but you must use the same symbol to open and close a string constant:

```
"55 Baker St."
'(212) 555-1212'
```

A quoted constant with no characters is a null string.

```
""

''
```

To combine strings and variable values in a quoted constant, you must use the colon preprocessor.

```
"The amount is :total"
```

A quoted constant that uses any special characters — colon, quote symbols, or backslash — must precede the special character with a backslash.

To use a colon as a character in a string expression (rather than as the colon preprocessor symbol), immediately precede the colon with a backslash or another colon, or put a blank space after the colon.

```
"\:"
"Do the following: 1)Type value; 2)Press EXIT."
```

You should also use the backslash if you wish to embed quote symbols in a constant. The other option is to use one symbol in the expression, and the other to enclose the constant.

```
"The message for :date is \":message\""
'The message for :date is ":message"'
```

Quoted constants are widely used in JPL, especially in `cat`, `msg`, and invocation statements.

Unquoted string constants

An unquoted string constant contains one or more non–blank, non–quote characters. When JPL evaluates the arguments of an invocation statement (`call`, `jpl`, `system`), a **JAM/DB*i*** statement (`dbms`, `sql`), a `public` statement, or a `load` statement, it passes an unquoted constant as a literal. Therefore, JPL never performs any variable substitution when it executes these eight statements. In fact, the only way to pass the value of a variable (rather than the variable's name) is to use the colon preprocessor in these statements:

```
jpl sales  7 today :total
```

`sales` is the name of a JPL procedure. `7` and `today` are unquoted string constants. `:total` represents a variable which the colon preprocessor will evaluate.

While unquoted constants contain any combination of letters and digits, their data type is always string. JPL will convert the constant's data type in the invoked function or procedure if it is necessary, and according to context.

6.3
OPERATORS

JPL supports all the operators available in the 'math window' in the Screen Editor, as well three integer bitwise operators. The table below summarizes each operator, its operands, and the data type of the value after the operation. Associativity is left to right, except for exponentiation where it is right to left.

Type	*Operator*	*Operation*	*Precedence*	*Result*
string	`()`	substring specifier	1	string
numeric	`@sum`	array sum	1	numeric
	`@date`	date calculation	1	numeric
	^	exponentiation	3	numeric
	/	division	5	numeric
	*	multiplication	5	numeric
	+	addition	6	numeric
	-	subtraction	6	numeric

Type	*Operator*	*Operation*	*Precedence*	*Result*
relational	>	greater than	7	logical
	>=	greater than or equal to	7	logical
	<	less than	7	logical
	<=	less than or equal to	7	logical
	==	equal to	8	logical
	!=	not equal to	8	logical
bitwise or integer	~	one's complement	4	integer
	&	bitwise and	9	integer
	|	bitwise or	11	integer
logical	!	not	4	logical
	&&	conjunction (and)	10	logical
	||	disjunction (or)	12	logical
assignment	=	equals	13	numeric

The use of operators will cause JPL to convert values to their appropriate types. There is no type conversion when the substring specifier is used. When the numeric operators are used, JPL must be able to convert the operands to numerics. When the bitwise operators are used, JPL must be able to convert the operands to integers; if the operands are numeric, JPL will truncate them to integer values. When the operators are relational, JPL must be able to convert the operands to one data type. When the operators are logical, JPL converts the operands to logical. The chart below describes the conversion of operands for relational and logical operators.

	String	*Numeric*	*Integer*	*Logical*
String	string	error	error	logical[1]
Numeric	error	numeric	numeric	logical[2]
Integer	error	numeric	integer	logical[2]
Logical	logical[1]	logical[2]	logical[2]	logical

1. a string is a logical true if it begins with the value of SM_YES .
2. a numeric or integer is a logical true if it is non–zero.

Below we discuss the operators that need further explanation, for example the substring specifier, `@date` and `@sum`. In case you have not used programming languages like C, we also discuss the bitwise operators. In JPL, they are particularly useful if you are using flags or masks. You should also see the *Programmer's Guide* for more information.

6.3.1

Substring Specifier

A substring specifier allows you to reference a part of any string. It may follow any variable name in the statement, and it specifies the beginning and the length of the substring. The syntax is

variable **(*m, n*)**

where

- variable is the name of a JPL variable, field, or LDB entry
- ***m*** is an integer expression whose value will be the beginning position of the substring, counting from 1
- ***n*** is an integer expression whose value will be the length of the substring.

A value for ***m*** is required. If ***n*** is not given, JPL assumes the end of the string as the default. ***m*** and ***n*** are integer values, and may be represented by integer constants or variables. ***m*** and ***n*** may be any value between 1 and the variable size (255 by default). If you reference a part of the variable beyond the last byte, those characters will not be available.

Substring specifiers are often used in string assignment statements like `cat` and in statements using logical expressions.

- For example, to extract a country code from an international phone number,

```
if int_phone(1,3) == "039"
  cat country "Italy"
```

- To find the first blank in a string,

```
for i = 1 while string(i, 1) != " " step 1
{ }
```

- To append a zip code extension,

```
cat zip(6) "-" extension
```

6.3.2
@date and @sum Operators

The numeric operators @date and @sum are also available in the 'math and checkdigit' window under field edits, in the Screen Editor. Since we explain them fully in the *Author's Guide*, we explain them only briefly here.

When using @date, the operand must be a field with a date format, or a literal date enclosed in parentheses, in the format specified in the message file entry SM_CALC_DATE (%m/%d/%4y or MON/DATE/YR4 by default). The @date operator will allow you to compare dates and perform arithmetic on dates.

- For example, if field1 and field2 have date edits, the statement,

  ```
  math field2 = @date(field1) + 30
  ```

  will set the date in field2 to 30 days past the date in field1.

- If days is a variable, and today is a field containing the current date, the statement,

  ```
  math days = @date(12/25/1990) - @date(today)
  ```

  sets days to the number of days until Christmas.

The @sum operator calculates the sum of all the non–blank occurrences in an array.

- For example, if quantities is an array and total is a field, the statement,

  ```
  math total = @sum(quantities)
  ```

  will put the sum of all occurrences of the array quantities in the field total.

6.3.3

Bitwise Operators

To accommodate bit manipulations, JPL provides three bitwise operators — bitwise AND (&), bitwise OR (|), and one's complement (~). A simple and useful application of a bitwise operator might use bitwise AND on a field's flag argument. All the flags are defined as hexadecimal values in the `include` files. We summarize them below. Since JPL does not recognize hexadecimals, we included their converted value in the last column.

Field Flags	*Defined in the* `include` *file:*	*hex*	*int*
VALIDED	smvalids	0x20	32
MDT	smvalids	0x40	64
K_ENTRY	sminstfn	0x0080	128
K_EXIT	sminstfn	0x0010	16
K_EXPOSE	sminstfn	0x0100	256
K_KEYS	sminstfn	0x0007	7
K_NORMAL		0	0
K_BACKTAB		1	1
K_ARROW		2	2
K_SVAL		3	3
K_USER		4	4
K_OTHER		5	5

Screen Flags	*Defined in the* `include` *file:*	*hex*	*int*
K_ENTRY	sminstfn	0x0080	128
K_EXIT	sminstfn	0x0010	16
K_EXPOSE	sminstfn	0x0100	256

Group Flags	*Defined in the* `include` *file:*	*hex*	*int*
K_ENTRY	sminstfn	0x0080	128
K_EXIT	sminstfn	0x0010	16
K_KEYS (see Field Flags)	sminstfn	0x0007	7

For example, if you are using the same function for field entry and exit, you might write a module like this:

```
vars ENTRY EXIT
cat ENTRY "128"
cat EXIT "16"

proc field_func
parms number data occ flags
if flags & ENTRY
  jpl do_process
else if flags & EXIT
  jpl do_exit_process
return
```

K_KEYS indicates why a function was called. See the following:

```
vars KEYS
vars NORMAL BACKTAB ARROW
cat KEYS "7"
cat NORMAL "0"
cat BACKTAB "1"
cat ARROW "2"

proc field_func2
parms num dat occ flg
if (flg & KEYS) == NORMAL
  return
else if (flg & KEYS) == ARROW
  msg emsg "Please use the tab key to move between fields."
return
```

6.4
EXPRESSIONS

An expression produces a new value by combining constants, variables, and operators. JPL evaluates an expression as one of the four data types, based on context. In all statements, JAM's colon preprocessor evaluates colon–expanded variables. In all expressions, JPL's statement processor replaces variable names with values.

String Expressions

A string expression combines one or more quoted string constants or values of string variables. The substring specifier is the only string operator.

```
'Montreal'
:district
"Processed :i items"
"Total cost is " total " for :x items"
phone (1, 3)
```

Numeric Expressions

A numeric expression combines variables and numeric constants with one or more of the numeric operators. You may use parentheses to control the flow of operations.

```
y + z
@sum(quantities)
x^y + y * (z^3/4 + 1) - x/2
86
```

Bitwise Expressions

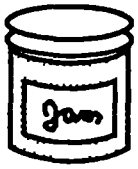
A bitwise expression uses variables or constants which have the data type integer, and any of the bitwise operators.

```
flag1 & flag2
x | mask
```

Logical Expressions

A logical expression uses the logical and relational operators to evaluate variables, numeric constants, integer constants, string expressions, numeric expressions, or integer expressions. The one restriction is the operands have the same type, or that they can be converted to the same type. For example, if you wish to compare a variable (or expression) with a numeric literal, JPL must be able to evaluate the value of the variable or expression as a numeric. If it cannot it will display an error message.

```
y
x != 7
(total * (1 + tax)) <= max_value
flag > ~flag
```



Chapter 7
Statements and JPL Commands

A statement consists of a command followed by zero or more arguments. In JPL, only one statement is permitted per line, but a statement may be continued to subsequent lines with a backslash (\). There is no maximum size for a JPL statement.

Braces mark the beginning and end of a block. Each brace must have its own line. The one exception is the null statement. In a null statement, both braces are permitted on the same line. A line beginning with a pound sign (#) is treated as a comment, and is ignored by the compiler. The `proc` command marks the beginning of a named JPL procedure.

Sample Statement	*Description*
`math total = subtotal * (1 + tax)`	assignment statement

`cat title \`
`  first_name \`
`  " " \`
`  last_name`
	statement continued with backslashes

`# calculate the total`	comment statement
`proc title`	proc statement
`for i = 1 while str(i,1) != " " step 1` `  { }`	null statement

Sample Statement	*Description*

`vars k`
`msg query /`
`"Do you want to do a widget?" k`
`while k`
`{`
`  jpl do_widget`
`  msg query /`
`  "Do you want to do another widget?" k`
`}`
	
	open block statement
	
	close block statement

7.1

SUMMARY OF JPL COMMANDS

Below is a summary of the JPL commands. If you are new to JPL, use this summary to get started. Begin with the assignment and definition commands; then move on to the commands for using loops, text, and procedure structures. Unless you are using application code, you can skip all the call commands except for `jpl`. When you are comfortable writing procedures, review the miscellaneous commands and the additional call commands.

definition and declaration:

`vars`	define a JPL variable in a procedure
`parms`	define a JPL variable if it does not exist; declare variable as a parameter
`retvar`	declare a variable to hold a return value

assignments:

`cat`	do string manipulation and assign value
`length`	count number of characters in a string and assign value
`math`	do numeric calculations and assign value

loops:

`break`	exit prematurely from a loop
`else`	execute statements(s) if the preceding `'if'` or `'else if'` fails

`else if`	conditionally execute statement(s) if the preceding `'if'` or `'else if'` fails
`for`	execute an indexed loop
`if`	conditionally execute statement(s)
`next`	skip to next iteration of loop
`while`	repeatedly execute statement(s) while a condition is true

text display:

`flush`	flush buffered output to the display
`msg`	display a message to the end–user

procedure structure:

`proc`	begin a named JPL procedure
`{ }`	null statement
`{`	begin statement block
`}`	end statement block

calls:

`atch`	execute a function attached to a field
`call`	execute a control string or prototyped function
`jpl`	execute a JPL function
`shell`	execute a system call and wait for user acknowledgement
`system`	execute a system call

miscellaneous:

`return`	exit from a JPL routine
`load`	read a JPL module into memory (module name is visible)
`public`	read a JPL module into memory (procedure names are visible)
`unload`	release memory used to hold load or public modules
`#`	begin a comment

JAM/DB*i* only:

`dbms`	execute a **JAM/DB*i*** directive
`sql`	submit a native dialect sql statement to the DBMS

You must begin every JPL statement with one of these commands. If you do not, **JAM** displays an error message for the unrecognized command. JPL created in the screen binary will display the message, if necessary, when you press XMIT to save the JPL procedures window.

7.2
REFERENCE

The following sections describe the commands in detail. Each description uses this structure:

- The command name.
- Usage synopsis, where

`{x}`	indicates an optional element, ***x***; the brackets should not be typed.
x...	indicates the element may be repeated one or more times.
`literal`	indicates a words to be typed verbatim; includes examples and literal entries.
italics	indicates screen names, file names, and variables; replace them with the appropriate values for your application.

- A full description of the command, with an explanation of its parameters, outputs, and actions.
- One or more examples of JPL statements or procedures demonstrating how the command is used.

The commands are listed in alphabetical order.

atch

execute a field function

SYNOPSIS

`atch` ***function {arg}***

DESCRIPTION

This command executes a field function, which is a function attached to a field, via the function lists. It is a useful way of calling a function that requires information about the field being validated. It has one optional argument, ***arg***. Once colon expansion is performed, ***arg*** is equivalent to text actually typed into a field. JAM will not strip quotes from this text, and it will preserve any blanks embedded in the text. If JAM is processing a field entry, validation, or exit function, the function called with the `atch` statement receives the following arguments:

- ***number*** field number
- ***contents*** ***arg***, if given; else the contents of the field
- ***occurrence*** occurrence number
- ***flags*** K_USER (invoked by user program; VALIDED and MDT bits both 0). See the *Programmer's Guide* for more information about these flags, and see section 6.3.3 on page 42, Bitwise Operators, for information about using these flags in JPL.

If the `atch` statement is not called in a field function, it passes the arguments of the current field (the field the cursor is in) to the called function. If no fields are present, the field number and occurrence number will be zero and the contents will be ***arg***, if given, or else the null string.

If you have declared a return variable with the `retvar` command, the attached function's return value is stored there.

EXAMPLE

```
# Display negative occurrences in red.
parms num dat occ flag
if dat < 0
  atch printred
else
  return
int printred ();
```

`printred` must be installed in the function list (e.g., `funclist.c`). For example,

```
/* list of field functions                                    */
struct fnc_data ffuncs[] =
{
   {"printred", printred, 0, 0, 0, 0}
};

int fcount = sizeof (ffuncs) / sizeof (struct fnc_data);


int
printred (field_num, field_data, occurrence, val_mdt)

int field_num;
char *field_data;
int occurrence, val_mdt;

{
   int att;
   att = RED | BLINK;
   sm_o_achg (field_num, occurrence, att);
   return;
}
```

break

exit prematurely from a loop

SYNOPSIS

`break` ***{integer constant}***

DESCRIPTION

The `break` command terminates execution of one or more enclosing `while` or `for` loops, and resumes execution at the command immediately following the last aborted loop.

integer constant equals the number of loops to break. If you do not specify it, JPL uses 1 as the default. If ***integer constant*** is greater than the number of existing loops, the program breaks out of all loops.

EXAMPLE

```
# Concatenate address and execute function for 100 entries.
# If cities[i] is empty stop executing the loop.
#
vars i address total
for i = 1 while i <= 100 step 1
{
  if cities[i] == ""
      break
  cat address cities[i] ", " states[i] "  " zips[i]
  call do_process :address
}
math total = i - 1
msg emsg "Done!  " :total " addresses processed."
```

SEE ALSO

`for, next, while`

call

execute a control string or prototyped function

SYNOPSIS — unprototyped functions

`call` ***function {text}***

SYNOPSIS — prototyped functions

`call` ***function {arg ...}***

DESCRIPTION

This command executes a built–in **JAM** function, or a function installed in the function list. You may install both library functions and your own application code functions in the function list, either in the list of control string functions or in the list of prototyped functions.

The use of prototypes affects how a `call` statement passes arguments to a function. The installation of functions and the use of prototypes is discussed in detail in Chapter 8, Calling Library Functions and Application Code.

If the function is not prototyped, the called function receives a character string beginning with the function name followed by any text. However, if the function is prototyped, **JAM** parses the string following the function name into space–separated arguments. Note that hex, binary or octal numbers may be passed to prototyped functions (see page 94). Single or double quotation marks must be used if ***arg*** has embedded blanks.

text and ***arg*** are colon expanded before `call` is executed.

If you have declared a return variable with the `retvar` command, JPL stores the function's return value in the variable. If you pass the names of variables to a prototyped function, you may use library functions to change the contents of the variable. For example, if you pass a field name to a prototyped function, the function can change the field's contents by using `sm_putfield`. See the **JAM** *Programmer's Guide* for details.

EXAMPLE

```
# Call a control string function to move the cursor
# to the first field on the screen.
call home

# Call a control string function to clear the array "stocks".
```

```
call sm_n_clear_array stocks

# Call a prototyped function to unprotect the field "accnum".
vars ans
msg query "Do you wish to change the account number?" ans
if ans
  call unprotect accnum
return

# Call a prototyped function to scroll to the specified
# occurrence in the array "names".
parms number
call sm_scroll names :number

# Call a prototyped library routine to change the value
# of a cursor key, using another prototyped library routine
# to access the values of key menmonics.

vars ret oldkey newkey
retvar ret
call sm_key_integer "NL"
cat oldkey ret
call sm_key_integer "TAB"
cat newkey ret
call sm_keyoption :oldkey 2 :newkey
```

To call library functions from JPL, install the name of the function in a function list. (If you wish to call your own C functions, you must also compile and link the function according to the directions in the JAM *Programmer's Guide*.) Below are sample function lists for the examples above. JPL calls the quoted function name.

```
/* list of control string functions
struct fnc_dat cfuncs[] =
{
  {"home", sm_home, 0, 0, 0, 0}
  {"sm_n_clear_array", sm_n_clear_array, 0, 0, 0, 0}

/* list of prototyped functions
struct fnc_dat pfuncs[] =
{
  {"unprotect(s)", sm_n_unprotect, 0, 0, 0, 0}
  {"sm_scroll(i,i)", sm_ascroll, 0, 0, 0, 0}
};
```

cat

concatenate and assign strings

SYNOPSIS

```
cat variable {string expression...}
cat variable (substring-specifier) {string expression...}
```

DESCRIPTION

The `cat` command concatenates one or more string expressions, and copies them into ***variable***. ***string expression*** is any string value (quoted constant or variable value). You may specify a character position where concatenation will begin or end (or both) by placing the values in parentheses after ***variable***. If you wish to append to the contents of ***variable***, include the name of ***variable*** in the list of expressions. If you do not give any arguments, the command clears the contents of ***variable*** (or the part of ***variable*** described by ***substring specifier***). You may also use substring specifiers on any ***string expression*** which is a variable or a colon–expanded variable.

A substring specifier allows you to reference a part of any string. It may follow any variable name in the statement, and it specifies the beginning and the length of the substring. The syntax is

variable (***m***, ***n***)

where

- ***variable*** is the name of a JPL variable, field, or LDB entry
- ***m*** is an integer value which is the beginning position of the substring, counting from 1
- ***n*** is an integer value which is the length of the substring.

A value for ***m*** is required. If ***n*** is not specified, JPL assumes the end of the string as the default. ***m*** and ***n*** are integer values, and may be represented by integer values between 1 and the variable's size (255 maximum). JPL will ignore any substring specifications beyond a variable's last byte.

EXAMPLE

```
# This is faster than math i = 1
cat i "1"

# This combines some field data with constants.
# Note that cat does not place blanks between items.
```

```
cat sons_name first " " last ", Jr."

# This is equivalent to the example above.
cat sons_name ":first :last, Jr."

# This tacks on a hyphen and a 4-digit extension to zip.
cat zip(6,5) "-" zip_extension

# This appends the extension by using zip as a source.
cat zip zip "-" zip_extension

# This clears a field.
cat zip
```

dbms

execute a **JAM/DB***i* directive

SYNOPSIS

dbms ***dbmstmt***

DESCRIPTION

The dbms command is only available with **JAM/DB***i*.

The ***dbmstmt*** is executed by **JAM/DB***i* after colon expansion and syntax checking. ***dbmstmts*** are typically directives that have no sql representation (ie., fetch next 10 rows), or directives that are not standardized across dialects of sql (ie., commit transaction).

EXAMPLE

```
# Fetch next set of rows
dbms continue

# Commit transaction
dbms commit
```

SEE ALSO

sql, **JAM/DB***i* Manual for your database.

else

execute commands if preceding 'if' or 'else if' fails

SYNOPSIS

```
else
single statement or block
```

DESCRIPTION

This command is valid only after an `if` or an `else if`. ***statement*** or ***block*** is executed only when the condition in the preceding `if` or `else if` statement is false. An `else` matches the last unmatched `if` in the same block.

EXAMPLE

```
# Beware of misplaced braces and ambiguous "elses."
# Examples 1 and 2 give the same results, but 3 does not.
#
# Example 1
if x == 1
if y == 2
   cat fld3 'yes'
else
   cat fld4 'no'

# Example 2
if x == 1
{
  if y == 2
    cat fld3 'yes'
  else
    cat fld4 'no'
}


# Example 3
if x == 1
{
  if y == 2
     cat fld3 'yes'
}
else
  cat fld4 'no'
```

SEE ALSO

`{ }` (block), `if`, `else if`

else if

execute commands if preceding 'if' or 'else if' fails

SYNOPSIS

`else if` ***logical expression***
single statement or block

DESCRIPTION

This command is valid only after an `if` or another `else if` statement. ***statement*** or ***block*** is executed only when both of these conditions are satisfied:

- the preceding if or else if statement failed, *and*
- the value of ***logical expression*** is true.

An `else if` matches the last unmatched `if` (or `else if`) in the same block.

Several conditions can be chained with `else if` commands.

EXAMPLE

```
#Determine a person's sex, based on his or her personal title.
if title == 'MR'
   cat sex 'Male'
else if title == 'MS'
   cat sex 'Female'
else if title == 'MRS'
   cat sex 'Female'
else if title == 'MISS'
   cat sex 'Female'
else
{
   cat sex 'Unknown'
   msg err_reset 'Please supply a title.'
}
```

SEE ALSO

`{ }` (block), `if`, `else`

for

execute an indexed loop

SYNOPSIS

```
for variable = numeric-expression \
   while logical-expression step arg
single statement or block
```

DESCRIPTION

This command provides an indexed loop. It has three clauses — the initial step, the loop condition, and the index step. These clauses control the repeated execution of the loop's statement or block.

The sequence of the command is as follows:

1. Initialize the index ***variable*** to the value of ***numeric expression.***
2. Evaluate ***arg.***
3. Evaluate ***logical expression.***
   - If false, stop execution of loop.
   - If true,

     a. execute statement or block

     b. increment index ***variable*** (***variable*** + ***arg***)

     c. return to step 3 (Evaluate ***logical expression***).

variable is an index variable. JPL increments it for each iteration of the loop. Its value is often used to keep a count or to process sequential array occurrences.

When the value of ***logical expression*** is false, JPL stops executing the loop. In the simplest case, it usually compares the index ***variable*** to some value that equals the maximum number of times that JPL executes the loop. It may use other values to determine the end of the loop. For example, you might use the index ***variable*** to evaluate array occurrences, but use the value of an occurrence, like a null string, to the end the loop. You may also test several conditions by using the logical "and" (&&) and "or" (||) operators.

arg is a constant or a variable. It has a positive or negative numeric value. Each time JPL executes the loop, it increments the index ***variable*** with the value of ***arg.*** JPL evaluates ***arg*** only once, before the first evaluation of ***logical expression.*** Therefore, if ***arg***

is assigned another value while the loop is executing, the new value will not affect the `step` process.

EXAMPLE

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

SEE ALSO

`{ }` (block), `while`

flush

flush buffered output to the display

SYNOPSIS

```
flush
```

DESCRIPTION

This command calls the library function `sm_flush`.

Since JAM uses a delayed–write feature, JAM does not immediately display output from `cat` and `msg` statements. Instead, it uses the output to update the screen image in memory. When the keyboard is opened (or the `flush` routine is called), JAM updates the display from this image.

You can use `flush` to force JAM to update immediately the physical display from the image in memory. Since JAM will update the display when it is waiting for keyboard input, use `flush` when JAM is not waiting for keyboard input. The `flush` command is useful when your procedure requires timed output or non–interactive display, for instance updating a time field.

EXAMPLE

```
# If this procedure is called as a screen entry function,
# it will print the banner one character at a time in field 1
# when the screen is opened.
proc welcome
vars w i
cat w "-------Sam's Discount Rentals-------"
for i = 1 while w(i,1) != "" step 1
{
    cat #1(i) w(i,1)
    flush
    jpl delay
}

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

if

conditionally execute statements

SYNOPSIS

`if` ***logical expression***
single statement or block

DESCRIPTION

This command provides for the conditional execution of other JPL commands. If ***logical expression*** is true, the following statement or block is executed. ***statement*** or ***block*** may be followed by an `else if` or an `else` statement.

EXAMPLE

```
# Supply a default value for an empty field
if amount == ""
   cat amount "N/A"

# Test a numeric value
vars x
math x = #5 - #4
# if x is non-zero, assign a value to recfld
if x
  cat recfld scrfld

# Test a string
vars more
msg query 'Would you like to see another?' more
if more
  return 0
else
  return 1
```

SEE ALSO

{ } (block), `else`, `else if`

jpl

execute a JPL routine

SYNOPSIS

`jpl` ***function {arg...}***

DESCRIPTION

This command calls a JPL procedure. A `jpl` statement passes the value of its arguments to parameters in ***function.***

In the statement, ***function*** may be either a procedure name, a module name, or the value of a colon expanded variable. JAM searches for ***function*** in the following places, in the following order:

1. `proc` names in the same JPL module
2. `proc` names in the active screen module
3. `proc` names put in memory by a `public` statement
4. module names put in memory by a `load` statement
5. module names in the memory–resident list
6. module names in the open form library
7. module names in the current directory
8. module names in the directories listed in `initcrt()` and the `SMPATH` setup variable Please refer to section 3.4 on page 14, Calling JPL Procedures and Modules, for more detailed information.

arg is an unquoted string constant, a quoted string constant, or the value of a colon–expanded variable. After colon expansion, the arguments are parsed into space–separated strings and passed to a `parms` statement in the called procedure, ***function.*** A quoted string constant is passed as a single argument with the quotes stripped off. If the value of colon expanded variable is string with one or more embedded spaces, the spaces in the string will be treated as delimiters between arguments. The total number of arguments, after colon expansion, should not be greater than twenty. If you try to pass more than twenty, the trailing arguments are ignored. They are not passed to the `parms` statement in ***function***, even if you have declared parameters for them. JAM will display an error message for a `parms` statement that declares more parameters than were actually passed.

EXAMPLE

```
# Loop through a set of synchronized arrays, calling a JPL
# routine to assemble an address from each "line," and a
# C subroutine to store the result in a file.
vars i r
retvar r
for i=1 while i <= 10 step 1
{
# The values of some the arguments being colon expanded may
# contain embedded blanks. If such an argument is not
# quoted, it will be separated into multiple arguments.
#
jpl getaddr whole ":name[i]" ":street[i]" \
  " :city[i]" :state[i] :zip[i]
if r > 0
       call store whole
}

proc getaddr
# Note the use of colon expansion for the first parameter.
# If this is not used, then only the local parameter receives
# the value of the cat statement, and no assignment will be
# made to "whole".
#
parms result_name name street city state zip
cat :result_name
if name == "" || street == "" || city == "" || state == ""
  return
cat :result_name name ', ' street ', ' /
  city ', ' state '  ' zip
return 1
```

SEE ALSO

load, parms, public, return, retvar, unload

length

count number of characters and make assignment

SYNOPSIS

`length` ***variable string–expression...***

DESCRIPTION

This command counts the number of characters in one or more ***string expression***s, and assigns the value to ***variable. string expression*** is a quoted string constant or the value of a variable (or a colon–expanded variable). You may use a substring specifier on any ***string–expression*** that is a variable.

`length` counts all characters and embedded blanks in ***string expression.*** Leading blanks in right–justified fields, and trailing blanks in left–justified fields are ignored. In quoted string constants, leading blanks are counted, but trailing blanks are ignored.

EXAMPLE

```
# Count the total number of characters in a customer's
# first and last name.
vars ln
length ln first last

# Count number of characters beyond the 100th byte.
vars c
length c address(100)
```

load

read a JPL module into memory

SYNOPSIS

`load` ***module...***

DESCRIPTION

This command reads one or more JPL modules into memory. ***module*** is either a file module, library module, or memory–resident module, or the value of a colon expanded variable representing one of these modules. Unless ***module*** has been compiled with `jpl2bin`, **JAM** will compile ***module*** when it executes the `load` statement. In either case, the execution of a `load` statement always converts the module (that is, converts it to the internal data structure).

Later calls to the module will not require any additional compilation or conversion. To execute a load module, call it with the statement

`jpl` ***module***

A load module has one entry point — its first, unnamed procedure. **JAM** begins execution at the top of the module, executing the first procedure, which must be unnamed. While the module may contain one or more named procedures, these procedures are accessible only to `jpl` calls within the load module. Since `load` makes only the module name visible, a load module may contain procedure names that could otherwise conflict with the names of other JPL modules or procedures. If you wish to use the named procedures as entry points to the module you should use the `public` command instead.

JAM will not load a module twice (or load two modules with the same name). If ***module*** is already in memory, any subsequent `load` statements will be ignored. You can release the memory used to hold a load module by using the `unload` command.

EXAMPLE

```
# Load three modules into memory for future use.
load validname defaultname blank
#
for i = 1 while i < 11 step 1
  jpl validname name[i]
#
# Note:  If validname had not been loaded, each call to the module
# would require reading the file off the disk, compiling it and
# converting it. The load statement avoids this repetitious processing
```

SEE ALSO

jpl, public, unload

math

do numeric calculations and make an assignment

SYNOPSIS

`math {%`***format}*** ***variable*** `=` ***numeric expression***

DESCRIPTION

The `math` command evaluates a numeric expression and assigns its value to ***variable.***

Numeric operators and expressions are discussed in Chapter 6. The operators are

+	add
–	subtract
*	multiply
/	divide
^	exponentiation
`@date`	date value
`@sum`	sum array

By default, JPL rounds the value of ***numeric expression*** to two decimal places. You may alter this with a format specifier to declare the total character length of destination and the number of decimal places. The syntax of ***format*** is:

m.n

where ***m*** and ***n*** are integer values (constants or variables). The value of ***m*** is the total number of characters, including digits, decimal place, and sign. If ***m*** is not given, ***m*** is assumed to be the size of the variable (maximum 255 bytes). The value of ***n*** is the number of digits after the decimal place.

You may also prevent JPL from rounding the value. Placing a `%t` before ***format*** tells JPL to truncate, rather than round, to the specified number of decimal places. If ***variable*** is a field or LDB entry, you may have already defined its floating point precision as a field edit. You may define a different precision in a `math` statement to override a field edit precision.

Multiple equations are permitted in the same statement. Use a semi– colon to separate assignments in the same statement. For example,

```
math i = i + 1; j = j + 2; k = k + i^j
```

EXAMPLE

```
# Calculate cost
math %9.4 total = @sum(checks)

# Compute the cost of an item
vars cost
math cost = (price * (1-discount)) * (1 + tax_rate)


# Compare values with different format specifiers
vars n
math n = 10 / 6
# Now n equals 1.67

math %6.4 n = 10 / 6
# Now n equals 1.6667

math %t6.4 n = 10 / 6
# Now n equals 1.6666

math %.0 n = 10 / 6
# Now n equals 2

math %t.0 n = 10 / 6
# Now n equals 1
```

SEE ALSO

cat

msg

display a message to the end–user

SYNOPSIS

```
msg query string (') {variable}
msg mode string–expression...
    where mode is one of the following:
        d_msg
        emsg
        err_reset
        qui_msg
        quiet
        setbkstat
```

DESCRIPTION

This command displays ***string–expression*** on the status line or in a pop–up window, in one of several modes. The modes correspond to a number of JAM library routines. ***string–expression*** is a quoted string constant or a variable (or colon–expanded variable). A `msg query` statement requires one and only one ***string*** as argument. A `msg` ***mode*** statement permits one or more ***string–expressions***. They are explained briefly below. See the *Programmer's Guide* for more details.

- `query` With this form of `msg`, you may display a question (***string***) to an end–user, and use his answer to determine how processing continues. JPL sets the optional ***variable*** to true or false by comparing the user's response to the message entries `SM_YES` and `SM_NO`. If the response is the value of `SM_YES`, ***variable*** is true; if it is the value of `SM_NO`, ***variable*** is false. Putting an exclamation point before ***variable*** reverses the logic (***variable*** is true if the user enters `SM_NO`'s value). If you do not specify a variable for ***variable***, JPL continues executing the procedure if the user enters the value of `SM_YES`, and exits immediately for the value of `SM_NO`. This may also be reversed with a single exclamation point after ***string***. The return value for both responses is zero.

- `d_msg` This mode displays one or more ***string–expressions*** on the status line and leaves it there, until cleared or replaced by another message. Text displayed using `d_msg` is buffered, and it will be displayed until you clear it (`msg d_msg ""`). It also may be temporarily replaced by a `msg` command with

another ***mode*** (except `setbkstat`). You can force JAM to update a `d_msg` message by using the flush command.

- `emsg` This mode displays ***string–expression*** as an error message; it is displayed until the end–user acknowledges it with a keystroke. The library routine `sm_option`, controls message acknowledgement. The message does not force on the cursor.

- `err_reset` This works like `emsg`, but it forces the cursor to be turned on at its current position.

- `qui_msg` This message displays ***string–expression*** as an error message until it is acknowledged. ***string–expression*** will be preceded by the `SM_ERROR` string from the message file, which is normally `ERROR:`. Cursor is not forced to be turned on.

- `quiet` Like `qui_msg`, but this message forces the cursor to be turned on at its current position.

- `setbkstat` This installs ***string–expression*** as the background status line. It will be displayed when no other message is active.

JAM provides several percent escapes for controlling the content and presentation of messages, and for specifying error message acknowledgment. They are:

- `%A` Change display attributes of message text.

- `%K` Display key label for logical key.

- `%B` Beep the terminal.

- `%N` Use a carriage return at this position in the error message text, and display the message in a pop–up window.

- `%W` Display the error message in a pop–up window, rather than on the status line.

- `%Md` Specify that user must press message–acknowledgment key to clear error message.

- `%Mu` Permit any keypress to serve as both error acknowledgment and data entry.

Percent escapes must be enclosed in quote characters in a JPL statement. See the JAM *Configuration Guide*, section 3.5 "Embedding Attributes and Key Names in Messages" for a complete description of each percent escape. In particular, `%A` has a table of hexadecimal values for setting display attributes and `%K` must be followed by a logical key mnemonic or hex value.

EXAMPLE

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

# Indicate that the current entry is being processed.
# Note that d_msg overrides delayed write and immediately
# flushes text to the screen.
msg d_msg 'Processing :name'
# Ask whether the user wants to quit the current screen.
vars quit
msg query 'Are you ready to quit?' quit
if quit
  return 0

vars field1 message
cat field1 "message"
cat message "Quick brown fox"

# This will display 'message' on the status line.
msg emsg field1
#
#This will also display 'message'.
msg emsg ":field1"
#
#This will display 'field1'.
msg emsg "field1"
#
#This will display 'Quick brown fox'.
msg emsg :field1


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

msg emsg "The menu toggle is %KMTGL"
msg query "Do you wish to continue? %B"
msg emsg "Enter value.%NPress XMIT."
msg qui_msg "%WInvalid password."
msg err_reset "%MdPlease enter a positive value."
```

next

skip to the next iteration of a loop

SYNOPSIS

```
next
```

DESCRIPTION

This command is valid only within the body of a `for` or `while` loop. It causes the current iteration of the loop to end, and the next iteration to begin. When a `next` statement is executed, all the statements between the `next` statement and the end of the loop are skipped. Control returns to the step, which increments the loop's index. Normal loop processing continues — test the condition, and execute the body of the loop if the condition is true. The `next` command applies only to the innermost enclosing loop.

`next` resembles the `continue` statement in C.

EXAMPLE

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

SEE ALSO

`break, for, while`

parms

declare parameters in a called JPL procedure

SYNOPSIS

```
parms variable ...
parms variable (size in bytes) ...
vars variable [number of occurrences] {(size in bytes)}...
parms variable [occurrence number] ...
```

DESCRIPTION

This command declares one or more parameters in a JPL procedure. Each argument which is passed by a `jpl` statement must have a parameter variable in the called function, if the function is to use those arguments.

Variables must be defined before they are used. When a variable is defined, storage is allocated for it. Sometimes it necessary to declare how a variable will be used. If a variable is to used as a parameter variable or a return variable it must be declared.

A `parms` statement declares one or more variables as parameters. If any of the variables has not been previously defined, this statement will also allocate storage for the variable. If you change the size of an existing variable, this statement will redefine the variable. If you are using array occurrences as parameters, first define the array with a `vars` statement, then use ***variable*** with an occurrence number as a parameter. For example,

```
vars x[3](10)
parms x[1] x[2] x[3]
```

After making the definition and/or declaration, **JAM** assigns the value of the associated argument to the parameter. One to twenty parameters may be declared in a single `parms` statement. Use a blank space to separate two or more variables on the same line.

If you declare more parameters than were actually passed, **JAM** will display an error message. (A `jpl` statement will pass up to twenty arguments.) If you declare fewer, the undeclared parameters will be inaccessible.

You may create global parameters for all procedures in the module by including the `parms` statements in the unnamed procedure at the top of the module. It is an error to declare parameters in the first, unnamed procedure of a screen or public module. Remember that **JAM** automatically passes arguments to field, group, and screen functions. (See section 3.4 on page 14, Calling JPL Procedures). Other global parameters

receive their values from the arguments named in the `jpl` statement that called the module.

EXAMPLE

```
jpl calculate :subtotal :state


proc calculate
parms amt st
#
if st == 'CA'
  cat tax '0.0725'
else if st == 'NY'
  cat tax '0.085'
else
  cat tax '0.00'
math total = amt * (1 + tax)
```

SEE ALSO

`vars, public`

proc

mark the beginning of a JPL procedure

SYNOPSIS

`proc` ***procedure***

DESCRIPTION

If your JPL module contains more than one procedure, you must begin the second and all subsequent procedures with a `proc` statement. These named procedures are executed when called by the statement `jpl` ***procedure***.

procedure is a character string which should uniquely identify the JPL function. It may be any length and contain any keyboard character except a blank space. When naming procedures in screen and public modules, be sure that the procedure names do not conflict with one another or with the names of load, memory–resident, library, or file modules that you are calling.

A module's first function is usually not named. When you call a file, load, memory–resident, or library module, JAM executes all the statements before the module's first `proc` statement. If you are calling any of these modules, the first procedure must be unnamed, or JAM will never execute the module. In field, screen, and public modules, the first unnamed procedure in the module is an auto function — that is, a function JAM executes without an explicit call. A field module's auto function is executed when JAM validates the field. If you name the first function, the module will not be executed. A screen module's auto function is executed when JAM opens the screen. If you name the function, the module has no auto function or global variables. The function, however, may be called with a `jpl` statement while the screen is displayed on the top–level. A public module's auto function is executed when JAM executes the `public` statement. If the first procedure is named, the module has no auto function or global variables, but the procedure may be called with a `jpl` statement while the module is in memory.

All variables defined with `vars` (or `parms`) statements in a unnamed first procedure are global to the module. Variables defined within a procedure are local to the procedure. A procedure may return values by making assignments to global variables or by using a `return` statement. Since a `proc` statement marks the end of one procedure, and the beginning of another, JPL procedures cannot be nested. `proc` statements are particularly useful in the screen module and in public modules. A screen module's procedures are available when the screen is active (displayed on the top–level). A public module's procedures are available until you `unload` the module.

EXAMPLE

```
#  This procedure is unnamed,
#  and it begins the module.
vars x y
cat x '100'
cat y '500'
#
#  This is the first named procedure.
proc sum
vars total
math total = x + y
msg emsg 'Total = :total'
```

SEE ALSO

jpl

public

read JPL modules into memory

SYNOPSIS

```
public module...
```

DESCRIPTION

This command reads the procedures contained in one or more JPL modules, compiles them if necessary, converts them to the internal data structure, and puts them in memory. It also executes the first procedure if it is unnamed. All procedures beginning with a `proc` statement are available throughout an application, or until you release the memory by listing ***module*** in an `unload` statement. ***module*** is either a file module, library module, memory–resident module or the value of a colon–expanded variable.

A `public` statement in an application makes all the `proc` names in a file visible for calling. Until you unload the module, every named procedure in the module is a possible entry point to the module. The name of the ***module***, however, is an entry point to the file, memory–resident, or library module, not the public module. Calling ***module*** with the statement "`jpl` ***module***" executes ***module***, unless ***module*** contains a procedure with the same name.

JPL will not make a module public more than once. If ***module*** is already public, subsequent `public` statements will be ignored.

EXAMPLE

```
# THIS IS A PROCEDURE IN A FILE MODULE NAMED 'ROUTINES'
proc quit
vars ans
msg query "Are you ready to quit?" ans
if ans
  return 1
else
  return 0

proc end
msg emsg 'Thank you. Have a nice day.'

# THIS IS A PROCEDURE AT THE TOP OF A SCREEN MODULE.
vars code1 code2
cat code1 '00090'
cat code2 '77654'
public routines
```

These functions might be called by a control string on the screen. For example, XMIT might have the following control string:

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

SEE ALSO

jpl, proc, load

return

exit from a JPL procedure

SYNOPSIS

return *{arg}*

DESCRIPTION

This command causes a JPL procedure to exit. Control is returned to the procedure that called it, if any, or to the **JAM** runtime system.

arg is an integer value represented by an integer constant or variable. JPL returns the value to the calling procedure. If you do not supply ***arg***, it returns the value 0.

Reaching the end of a JPL procedure or file causes an automatic return. Use the `return` statement to exit before the end of a procedure, or to return a value other than zero. Return values are used in procedures executed as field functions, and in control string functions with target lists, and in JPL procedures with declared `retvars`.

EXAMPLE

```
# Call procedure checknum to evaluate value of a field called
# num. Based on its value return an integer that will
# determine the next procedure to be called.
#
retvar r
jpl checknum
if r == 1
  jpl lownum_process
else if r == 2
  jpl midnum_process
else
  jpl hinum_process


proc checknum
if num < 0
  return 1
else if num < 500
  return 2
else
  return 3
```

SEE ALSO

`retvar`

retvar

establish a variable to hold a return value

SYNOPSIS

```
retvar variable
```

DESCRIPTION

`retvar` declares a return ***variable***. ***variable*** must be defined before it can be declared as a return variable. Fields and LDB entries are valid variables. JPL variables are also valid, after they are defined by a `vars` or `parms` statement.

A procedure or function is called with one of the invocation statements (`atch`, `call`, `jpl`, or `system`). If you declare a `retvar` ***variable*** before calling the function or procedure, the value returned by the function or procedure will be assigned to ***variable***. For JAM/DB*i* users this also applies to `sql` and `dbms` statements. Once you declare it, ***variable*** serves as a return variable for the lifetime of the procedure or load module, or until it is redefined. Every invocation statement in the procedure or load module may use the same the return ***variable***. If ***variable*** is omitted, JPL clears the value of the previous `retvar` variable, and the variable no longer serve as a return variable for the rest of the procedure.

EXAMPLE

```
# Call procedure checknum to evaluate value of a field called
# num. Based on its value return an integer that will
# determine the next procedure to be called.
retvar r
jpl checknum
if r == 1
  jpl lownum_process
else if r == 2
  jpl midnum_process
else
  jpl hinum_process

proc checknum
if num < 0
  return 1
else if num < 500
  return 2
else
  return 3
```

SEE ALSO

`atch, call, dbms, jpl, return, sql, system`

shell

execute a system call & wait for user acknowledgement

SYNOPSIS

`shell` ***{command {arg...}}***

DESCRIPTION

command is the name of the command to be executed by the operating system. ***arg*** is an unquoted string constant, quoted string constant, or colon–expanded variable. When executed, the screen is cleared, and any program output displayed. A "Please hit space bar" message is displayed when the command finishes execution. When the message is acknowledged, the JAM screen is refreshed and screen processing resumes.

If you have established a return value variable with the `retvar` command, the exit status of the program will be available there.

The JPL command `system` performs the same functions, but does not display the acknowledgement message.

EXAMPLE

```
# On a UNIX system, check a directory listing.
shell ls -l
#open a file...
```

SEE ALSO

`retvar, system`

sql

submit a native dialect sql statement to the DBMS

SYNOPSIS

sql ***sqlstmt***

DESCRIPTION

The `sql` command is only available for **JAM/DB*i*** users.

The ***sqlstmt*** is executed by the DBMS, after colon expansion. **JAM** performs no other translation of ***sqlstmt***. **JAM/DB*i*** does not perform syntax checks on this argument.

EXAMPLE

```
# Retrieve all of Acme's pest removal products that cost less
# than "pricey" dollars.
sql select description, partnum, cost from prodtbl \
  where supplier = "Acme" \
  and type = "roadrunner" and cost < :pricey
```

SEE ALSO

`dbms`, **JAM/DB*i*** manual for your database.

system

execute a system call

SYNOPSIS

```
system program {arg...}
```

DESCRIPTION

program is the name of the program to be executed by the operating system. ***arg*** is an unquoted string constant, quoted string constant, or colon–expanded variable. When executed, the screen is cleared, and any program output displayed. When it exits, the JAM screen is refreshed and screen processing resumes.

If you have established a return value variable with the `retvar` command, the exit status of the program will be available there.

If wish to have a delay before returning to JAM, for example if the user needs to examine the program output before it is cleared, use the `shell` command instead.

EXAMPLE

```
# On a UNIX system, check whether a file exists.
vars status
retvar status
system test -f :filename
if !status
  return
#process the file...
```

SEE ALSO

`retvar`, `shell`

unload

free the memory holding load and public modules

SYNOPSIS

unload ***module***...

DESCRIPTION

This command releases the memory used to hold one or more JPL modules listed as arguments in a previous `load` or `public` statement. If you call ***module*** after unloading it, JAM will read it in from disk (or the memory–resident list, or open library). If you call any procedures from ***module*** after unloading, JAM will display an error message.

A module should not be unloaded while it is being executed.

EXAMPLE

```
# LOAD A FILE, CALL IT IN A LOOP,
# AND UNLOAD IT AFTER EXITING THE LOOP
load validname
for i = 1 while i < 11 step 1
  jpl validname name[i]
unload validname


# THIS PROCEDURE MIGHT BE CALLED BY A SCREEN EXIT FUNCTION
proc screen_exit
unload fld_routines validname calcs
```

SEE ALSO

`jpl, load, public`

vars

define JPL variables

SYNOPSIS

```
vars variable-name ...
vars variable-name [number of occurrences] (size) ...
vars variable-name [number of occurrences] ...
vars variable-name (size) ...
```

DESCRIPTION

This command creates one or more JPL variables. Variables defined within a procedure are local to the procedure. Variables defined outside any procedures in a JPL module are available to all procedures in the same module. JPL executes the global `vars` statements in screen and public modules when the module is activated (by opening the screen or executing a `public` statement).

Variable names are any combination of letters, digits, and underscores that does not begin with a digit. The special characters dollar sign and period are also permitted in variable names created with `vars` or `parms` statements. The maximum length of a variable name is 31 characters. Names which are longer than 31 characters will be truncated. Use a blank space to separate two or more variable names in the statement.

To define a variable as an array, place its number of occurrences in square brackets after its name. To specify the size of each occurrence, indicate, in parentheses, its number of characters and place it after any occurrence declaration. If the size is not declared, JPL will use the maximum variable size of 255 characters.

The value of a newly declared variable is the null string (`""`).

EXAMPLE

```
vars name(50) flag(1)
vars address[3](50) abbrevs[10]
vars i(5)
```

SEE ALSO

`parms`

while

repeatedly execute a block while a condition is true

SYNOPSIS

```
while logical expression
single statement or block
```

DESCRIPTION

The `while` statement repeatedly executes a single statement or block as long as the value of ***logical expression*** is true. JPL evaluates ***logical expression*** before each iteration of the loop.

EXAMPLE

```
vars k
msg query "Do you want to do a widget?" k
while k
{
  jpl do_widget
  msg query \
    'Do you want to do another widget?' k
}
```

SEE ALSO

{ } (block), `break`, `for`, `next`

# #

begin a comment statement

SYNOPSIS

#*text*

DESCRIPTION

A pound sign begins a comment statement. The JPL compiler ignores all lines beginning with the comment symbol. Any number of leading and trailing blanks are permitted with this symbol. A comment statement may not be continued with a backslash; each comment line must begin with a pound sign.

JPL also supports the comment symbol used in previous JAM releases, the colon (:). We recommend the use of #, however, since others reading your procedures may find the colon difficult to see, or confuse it with colon expansion. Use of the pound sign prevents these problems, and it is consistent with the comment notation in JAM files, like the video file.

Comments cannot be embedded in JPL statements. JAM uses an embedded pound sign as a reference to a field number, and it uses an embedded colon as the colon expansion character.

EXAMPLE

```
#comment
#  comment
```

{ }

mark beginning and end of statement block

SYNOPSIS

```
{
statement...
}
```

DESCRIPTION

Curly braces mark the beginning and end of statement blocks. Neither can be on the same line as another JPL command. Therefore, each brace is on its own line, with a left brace marking the beginning of the block, and a right brace marking the end. (One exception is the null statement.)

Blocks may be nested. A right brace matches the closest previous left brace.

EXAMPLE

```
# This loop executes 2 jpl procedures 10 times. When the loop
# ends, JPL displays the message "Done!"
#
for i = 1 while i <= 10 step 1
{
  jpl get_value :part[i]
  jpl do_routine :part[i]
}
msg emsg "Done!"
```

{ }
null statement

SYNOPSIS

```
{ }
```

DESCRIPTION

A left and right brace on the same line indicate a null statement. You might, for example, use the efficient `for` statement to keep a count while testing a condition, but not need to execute any other statements. In such a case, you could use a null statement after the `for` statement.

EXAMPLE

```
# This procedure finds the position of the first blank in a
# string. Using the index variable, i, as a substring
# specifier, it examines each character in str's value. Once
# it finds a blank, the loop ends and i equals the position of
# the blank character.
#
for i = 1 while str(i, 1) != " " step 1
  { }
```

Chapter 8
Using Library Functions and Application Code

JPL provides access to the JAM library functions and to your own application code. This feature greatly increases JPL's capabilities. You may develop sophisticated final applications, writing little or no application code. This chapter briefly describes how to make these library and application functions available to JPL procedures. Our examples use C syntax. If you are using another programming language, see the JAM *Programmer's Guide* for specific information.

8.1 FUNCTION LIST

If you wish to call JAM library functions, standard C functions, or your own C functions from a JPL procedure you must install the functions in a function list. JAM has five function lists. In JPL you may call functions from three of these—the field, control string, and prototyped function lists. Use an `atch` statement to execute a field function, or a `call` statement to execute a control string or prototyped function. The syntax of the function list is,

```
  /* list of field (entry/exit/validation) functions          */
struct fnc_data ffuncs[] =
{
  { "fdummy", fdummy, 0, 0, 0, 0 }
};
int fcount = sizeof (ffuncs) / sizeof (struct fnc_data);

  /* list of JAM control functions                            */
struct fnc_data cfuncs[] =
```

```
{
  { "cdummy", cdummy, 0, 0, 0, 0 }
};
int ccount = sizeof (cfuncs) / sizeof (struct fnc_data);

  /* list of prototyped functions                                    */
struct fnc_data pfuncs[] =
{
  { "pdummy", pdummy, 0, 0, 0, 0 }
};
int pcount = sizeof (pfuncs) / sizeof (struct fnc_data);
```

Using `pfuncs` as an example, the structure is defined:

"pdummy" The name used to call the function from your application; may be the same as the address of the function.

In addition to the function name, you may also list a prototype. The valid prototypes are discussed below. If you use any other prototype, JAM will display an error message for the faulty prototype.

pdummy address of the function (the name used to call the function from application code).

`0,0,0,0` language: 0 indicates C
intrn_use: installation parameter
appl_use: parameter for your use
reserved: must be 0

8.1.1
Prototypes

You may prototype, that is, specify the data types of a function's arguments. The prototype is enclosed in parentheses. It immediately follows the function name, and is inside the quotation marks. JAM permits the void prototype or one of the 21 integer and string prototypes. The prototypes include the most likely, and the most useful combination of arguments that can be managed by JAM. Only these prototypes are permitted. They are:

`()` This is the "void" prototype. Use it when you want to install in the prototyped list a function which has no arguments.

```
(i)
(s)

(1, i)
(s, i)
(i, s)
(s, s)

(i, i, i)
(s, i, i)
(i, s, i)
(s, s, i)
(i, i, s)
(s, i, s)
(i, s, s)
(s, s, s)

(i, i, i, i)
(s, i, i, i)
(s, s, i, i)
(s, s, s, i)
(s, s, s, s)

(i, i, i, i, i)

(i, i, i, i, i, i)
```

In addition to `i` and `s`, `str` and `int` are also permitted. We recommend that you use the abbreviations, however, to save space.

You may prototype any function in the function list. Keep in mind, however, that when JAM executes a call to a field function, it automatically passes four arguments (field number, contents, occurrence, flags) to the field function. If you prototype arguments in a field function, the customary arguments will not be passed, although they may be obtained via the library routine `sm_inquire`. Refer to the JAM *Programmer's Guide*.

Control string functions are called with a caret. JAM strips off the caret and passes the entire character string (the function name and any following text) as a single argument. If you have prototyped the function, JAM parses the string into space–separated arguments. To pass an argument which contains blank spaces, enclose the argument in quotes.

While you may give a prototype to any function in the function list, installing the function in the prototyped list has a distinct advantage — you may call a function installed in `pfuncs` from anywhere in your application. Depending on where the call is made,

JAM first searches the appropriate structure (`sfuncs`, `gfuncs`, `ffuncs`, or `cfuncs`) for the function. If it does not find the function, it searches `pfuncs`. You may also install an unprototyped function in `pfuncs`, and therefore take advantage of the increased scope without overriding default arguments.

A sample `pfuncs` with installed functions might appear as follows:

```
struct fnc_data pfuncs[] =
{
   {"n_putfield(s,s)", sm_n_putfield, 0, 0, 0, 0},
   {"getcurno", sm_getcurno, 0, 0, 0, 0}
};
```

The last function has no prototype. If a function in `pfuncs` has no prototype, JAM treats the function like a control string function. When JAM executes it, JAM passes it a copy of the string that invoked the function. This only works on the prototyped (and control) function list. If a function has the void prototype, `()`, then JAM does not pass any arguments to the function.

The *Programmer's Guide* explains installing functions in the function list with `sm_install`.

Hexadecimal, Octal and Binary Arguments in Prototyped Functions

Normally, hexadecimal, octal and binary numbers cannot be used in JPL. But when a JPL procedure calls a prototyped function that takes an integer argument, a string to integer conversion takes place. This conversion permits the use of hexadecimal, octal, or binary values as arguments. The format for these values is as follows:

Base	*Format*
Binary	0b***nnnn***
Octal	0***nnnn***
Hexadecimal	0x***nnnn***

For example, a prototyped function can be called from JPL with the syntax:

```
call sm_keyoption 0x10a 2 0x109
```

8.2 LIBRARY FUNCTIONS

The *Programmer's Guide* describes the JAM library functions. JAM provides routines for initialization, screen display, data entry, keyboard entry, cursor control, data access,

mass storage and retrieval, and message display, as well as routines to change the operation of other functions.

For example, to call the library function `sm_clear_array`, add the following to the list of prototyped functions, and recompile the function list:

```
{ "sm_clear_array(i)", sm_clear_array, 0, 0, 0, 0}
```

In the *Programmer's Guide*, the synopsis for the function `sm_clear_array` lists one argument, ***field_number***, with the type `int`. To pass the name of the array, use the function `sm_n_clear_array`, with a string prototype.

```
{ "sm_n_clear_array(s)", sm_n_clear_array, 0, 0, 0, 0}
```

In a JPL procedure, you could clear an array named ***total*** with this statement.

```
call sm_n_clear_array total
```

If you choose, you may call a function by a name other than the one used in the C program to reference the function (the function address). For example, with the installation of

```
{ "clr_array(s)", sm_n_clear_array, 0, 0, 0, 0}
```

you would call the library function with the following statement,

```
call clr_array total.
```

Arguments to a prototyped function may be enclosed in parentheses. For example,

```
call sm_n_clear_array (total)
```

is equivalent to

```
call sm_n_clear_array total
```

8.2.1

Acessing Key Mnemonics

Values defined in C header files are not accessible from JPL, but several JAM library routines require these values as arguments. The library routine `sm_key_integer` returns the integer value of a key mnemonic defined in `smkeys.h`. Rather than looking a value up manually, you can prototype `sm_key_integer` and call it from JPL to access key values. An example appears below.

```
vars ret key1 key2
retvar ret
call sm_key_integer "NL"
cat key1 ret
call sm_key_integer "TAB"
cat key2 ret
call sm_keyoption :key1 2 :key2
```

8.3
HOOK FUNCTIONS

Application code may also be called from JPL procedures. Your code must be compiled and linked according to the directions in the *Programmer's Guide*. The name of the function should then be installed with the utility `sm_install`, as it is for library functions. The program is called with the `call` verb and the name defined in the list of prototyped or control string functions. Again, the prototype should correspond to the arguments of the function.

8.4
BUILT–IN FUNCTIONS

The **JAM** built–in functions described in the *Programmer's Guide* may be called from JPL. For example, to simulate a keyboard entry of the string "JYACC", call the function `jm_keys`. No installation is necessary.

```
call jm_keys JYACC
```

Some built–in functions, such as `jm_exit` or `jm_goform`, are control flow functions, and you should avoid using them in some JPL procedures. In particular, we strongly discourage the use of `jm_exit` in screen and field JPL modules. The execution of this function often destroys data structures needed to complete the JPL procedure, and may cause the program to crash. The use of a target list with a control string function is the safest and most recommended way to alter control flow in an application.

Chapter 9
Performance Considerations

This chapter provides some performance considerations for JPL. We discuss methods for reducing or eliminating runtime JPL compilation. In addition, we give some suggestions for writing more efficient statements.

9.1
JPL COMPILATION

The text you enter in a file with a text editor, or in a JPL procedure window with the Screen Editor, creates the ASCII version of your JPL module. Before using the module, JAM must compile and convert it.

The compilation changes JPL keywords to tokens, partitions the module into procedures, and performs elementary syntax checks. During the syntax check, JAM will display error messages for invalid command words, or missing arguments. Since JAM performs colon preprocessing at runtime, a module is never fully compiled until it is executed. Compilation occurs automatically when you save a field or screen module (by hitting XMIT). You may use a separate utility, `jpl2bin`, to compile file modules. JAM, of course, processes a JPL module more efficiently if it is pre–compiled. You must compile a module with this utility before placing it in a library or memory– resident list.

To accommodate screens created with previous versions of JAM, JAM does not assume that a JPL module is compiled. When it initializes the module, JAM will automatically invoke the compiler for an uncompiled module. Because of differences in binary and text files on some platforms, JAM opens an uncompiled JPL file module twice — the first time in binary mode, the second time in text mode. JAM does not save the compiled version of a JPL file module. All subsequent calls to the module will require recompilation.

From the compiled module, JAM constructs internal data structures. We call this step "conversion." This is a fairly simple task in the initialization process, but it does require one memory allocation per JPL statement.

The `load` command (or the library function `sm_jplload`) compiles a module — if necessary, converts its, and keeps it in memory for later use. The `public` command (or the library function `sm_jplpublic`) similarly compiles, if necessary, and converts a module, and holds it in memory. `public`, however, makes every named procedure an entry point to the module. A load module has only one entry point, the first, unnamed procedure which is called by the module name. Therefore, all calls to a load module use the module name, and all calls to a public module use procedure names. See the reference section for more information.

9.1.1
Using `jpl2bin`

The utility `jpl2bin` converts the ASCII version of a JPL file to a binary one. The name of the binary file will be the same as the file module with a `.bin` extension. The utility is used at the system level with this command,

```
jpl2bin {-pv} {-eextension} filename ...
```

where

filename	is the name of a JPL file module.
`-p`	puts the binary file in the same directory as ***filename***.
`-v`	lists the name of each file as it is processed.
`-e`	appends ***extension*** rather than `bin` to the binary file name.

See the JAM *Utilities Guide* for a complete description.

9.1.2
Adding JPL to a Library

Once you convert a file with `jpl2bin`, you can add it to a library with the utility `formlib`. `formlib` is explained in the *Utilities Guide*.

9.1.3
Making JPL Memory–resident

To make a JPL module memory–resident, compile the file module with the utility `jpl2bin`. Convert the binary version to source language with the appropriate utility

— `bin2c`, for example. Add the names to the memory–resident list with `sm_formlist`. Link the JPL file with the application executable and recompile the executable. See the JAM *Programmer's Guide* for an explanation of `sm_formlist`.

9.2 MORE EFFICIENT STATEMENTS

You may make performance improvements in JPL by writing more efficient statements.

9.2.1 Initializations

You may minimize character and numeric conversions in initialization statements. The `cat` command is more efficient than the `math` command.

For example,

```
cat counter '1'
```

rather than

```
math counter = 1
```

9.2.2 Loops

Similarly, the increment in a `for` loop is more efficient than a `math` increment in a `while` loop.

```
for i = 1 while i < 10 step 1
  {
    ...
  }


while i < 10
  {
    ...

    math i = i + 1
  }
```

9.2.3

Colon Usage

Your use of colons in JPL procedures also affects performance. Before the JPL interpreter executes a JPL statement, the entire module is compiled and converted. Then, statement by statement, the colon preprocessor replaces colon–expanded variables with values, and passes the statement to the JPL interpreter which evaluates expressions and/or arguments and executes the statement. The colon preprocessor works by scanning a statement from right to left, looking for colons. When it encounters a colon, it checks for a left parenthesis before the colon. If there is none, it accumulates characters from left to right after the colon, stopping once it reaches a blank space or a character that cannot be expanded (like a quote character). This is the variable which the preprocessor will replace with its value. If there is a left parenthesis immediately before the colon, the preprocessor stops accumulating characters when it reaches a right parenthesis. If there is an occurrence number and/or substring specifier immediately after the variable name (no blank space or parentheses) the colon preprocessor will replace the specified occurrence and/or substring.

There are three ways to prevent colon expansion in JPL. Preceding a colon by another colon or by a backslash (\) prevents colon expansion. The other way is to place a space after the colon.

```
msg emsg "::tax"
msg emsg "\:tax"
msg emsg ": tax"
```

The last example is the most efficient way of preventing colon expansion, since JAM will not need to copy the argument to a buffer to remove the : or the \.

The messages will appear like this

```
:tax

:tax

: tax
```

You can use parentheses to simplify writing references that use colon expansion.

```
msg emsg :(city)[i]
```

Eliminating unnecessary parentheses reduces processing.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Programmer's Guide

TABLE OF CONTENTS

Chapter 1
Introduction

This document is intended for JAM Programmers. We discuss the development and creation of executable JAM programs incorporating the Screen Manager, developer–written hook functions, and the JAM Executive. We will briefly touch on how custom executives may be written. Finally, there is a comprehensive reference of JAM library functions.

Discussions on the creation of JAM screens, data dictionaries, and keysets are found in the *Author's Guide*. JPL is fully documented in the *JPL Guide*.

This document assumes that the reader has previously read the JAM Development Overview and the *Author's Guide*. The Development Overview is particularly important as the major architectural components of JAM are explained there in detail.

JAM is written in C, and the C programming interface and libraries are distributed with every license. For most of this document, we will discuss JAM programming from the perspective of a C programmer. Those who wish to program in other languages (e.g. Fortran, Cobol, PL/1) may obtain additional language interfaces on some platforms.

You will need to program in C (or some other supported third–generation language) to accomplish the following tasks:

- To customize JAM to your environment or application by modifying the main program provided in source form with the product.
- To write hook functions that do application–specific and back–end processing during the execution of the application.
- To take full control of the application by writing an application–spe-

cific executive[1].

- To create executable JAM Programs.

As discussed in detail in the Development Overview, JAM Applications consist of screens, a data dictionary, hook functions, and an executable program. The creation of screens and data dictionaries is discussed in the *Author's Guide*. JPL programming is discussed in the *JPL Guide*. In this chapter, we discuss how to create a JAM program. Compilation and linking are specific to platforms and operating systems and are discussed in the Installation Guide.

Two different versions of an application can be created with JAM. The Application Executable is the program delivered to the end–user to control the run time application. The JAM Authoring Executable is used to create application components and test the application during development. Only the JAM Authoring Executable will grant user access to the Screen Editor, the Data Dictionary Editor, and the Keyset Editor. The JAM Authoring Executable can only be used for the testing of applications that use the JAM Executive.

The JAM product is distributed with a plain version of the JAM Authoring Executable; one without any application–specific hook functions or data structures linked in. It is called `jxform`. Its use is detailed in the *Author's Guide*. New versions of the Authoring Executable with application–specific hook functions linked in may be created, but JAM licenses specifically forbid their distribution as runtime applications.

1.1
APPLICATION EXECUTABLE

Application Executable programs fall into two categories: those that use the JAM Executive to manage the flow of control from screen to screen, and those that use an application–specific executive. We discuss both of these approaches in the sections that follow.

1.1.1
Applications Using the JAM Executive

In applications that use the JAM Executive, most of the control flow is encapsulated in the screens. The majority of the C programming task is to write hook functions (see

1. It is strongly recommended that the JAM Executive be used in all but the most unusual of circumstances. A comparison of the JAM Executive with your own executive is presented in the Development Overview.

Chapter 2) that are called by the Screen Manager or by the JAM Executive when certain events occur.

Applications that use the JAM Executive will need to be linked with the Screen Manager library `sm`, the JAM Executive library `jm`, and, in general, the standard math library on your system.

JYACC provides the main routine source code for applications that use the JAM Executive in a file called `jmain.c`. This routine performs various necessary initializations before calling the function that starts up the JAM Executive. You may want to modify this code to change JAM's default behavior, see section 1.3 below.

1.1.2
Applications Using a Custom Executive

In rare cases, a developer may choose to write a custom executive, one that is specific to a particular application. In custom executives, no library functions specific to the JAM Executive should be used. The JAM Executive functions may only be used in applications using the JAM Executive — they are listed in section 12.15 on page 176.

Applications that do not use the JAM Executive should be linked with the Screen Manager library `sm` and, in general, the standard math library. If the LDB is needed, the JAM Executive library `jm` should also be linked in, but it is important the application not call any JAM Executive routines.

The main routine for a very simple application using a custom executive is shown below[2]. The application brings up a screen called `mainform` and allows a user to search an underlying database or update a record in that database. The user might also bring up a detail window named `mywindow` from which a report could be printed. Striking EXIT while `mywindow` is displayed will close it. Striking EXIT while `mainform` is displayed will terminate the application.

```
/* INCLUDE FILES */
#include "smdefs.h"
#include "smkeys.h"

/* FORWARD DECLARATIONS */
void show_my_window ( ) ;

/* EXTERNAL DECLARATIONS */
extern void do_my_update ( ) ;
extern int do_my_search ( ) ;
extern void do_my_print ( ) ;
```

2. Note that JPL is available to applications that do not use the JAM Executive. Note also that hook functions may be installed and used in applications that do not use the JAM Executive. These applications, however, will not be able to use control strings.

```
/* MAIN DECLARATION */
int
main ( argc, argv )
int argc ;
char **argv ;
{
        int the_key ;   /* Key pressed by user */

        /* INITIALIZE THE SCREEN MANAGER */
        sm_initcrt ( "" ) ;

        /* INSTALL HOOK FUNCTIONS */
        sm_do_uinstalls ( ) ;

        /* DISPLAY APPLICATION MAIN FORM */
        if ( sm_r_form ( "mainform" ) )
        {

                /* DISPLAY ERROR MESSAGE */
                sm_err_reset ( "Unable to display Main Form" ) ;
        }
        else
        {

                /* LET USER INTERACT WITH SCREEN */
                while ( ( the_key = sm_input ( IN_AUTO ) )
                    != EXIT )
                {
                        switch ( the_key )
                        {
                        case XMIT:
                                do_my_update ( ) ;
                                sm_cl_unprot ( ) ;
                                break ;

                        case PF1:
                                show_my_window ( ) ;
                                break ;

                        case PF4:
                                if ( do_my_search ( ) )
                                {
                                        sm_cl_unprot ( ) ;
                                }
                                break;
                        }
                }

        }

        /* RESET THE TERMINAL */
```

```
        sm_resetcrt ( ) ;

        /* EXIT PROGRAM */
        return ( 0 ) ;
}

void
show_my_window ( )
{
        int the_key;    /* Key pressed by user */

        /* DISPLAY THE WINDOW */
        if ( sm_r_at_cur ( "mywindow" ) )
        {

                /* DISPLAY ERROR MESSAGE */
                sm_err_reset ( "Unable to display My Window" ) ;
        }
        else
        {

                /* LET USER INTERACT WITH SCREEN */
                while ( ( the_key = sm_input ( IN_AUTO ) )
                    != EXIT )
                {
                        switch ( the_key )
                        {
                        case XMIT:
                                do_my_print ( ) ;
                                break ;
                        }
                }
                /* CLOSE THE WINDOW */
                sm_close_window ( ) ;
        }

        return ;
}
```

Screen Manager Initialization

After all the header files and declarations at the top of the source module, the Screen Manager and the terminal are first initialized with a call to `sm_initcrt`. Since an empty string is passed as the argument, the search path for screens is expected to be found in the environment.

Install Hook Functions

The function `sm_do_uinstalls` is defined in `funclist.c`, provided as source with JAM. It is used to install Screen Manager hook functions, described below in Chapter 2.

Display the Main Form

After initialization is complete in the main routine, the screen `mainform` is opened as a form with a call to `sm_r_form`. If an error occurs, the program will terminate.

Display My Window

In the function `show_my_window`, `mywindow` is displayed at the cursor position with a call to `sm_r_at_cur`.

Handle Errors

Error messages when screen display fails are placed on the status line with the call to `sm_err_reset`. This routine takes a single string argument, and places that string on the status line. The user is forced to acknowledge the error by striking the space bar[3].

Activate Screen

Both `mainform` and `mywindow` are activated within a loop. The loop terminates if the user strikes the EXIT key, which causes the routine `sm_input` to return with the return code EXIT defined in `smkeys.h`. The actual data entry, cursor movement, help processing, character edit masking, and validation are handled within `sm_input`, so the programmer need not be concerned with them. Whenever the user strikes TRANSMIT, EXIT, or some other function key, `sm_input` returns control to the calling program. In the main routine when `mainform` is displayed, the keys TRANSMIT, PF1 and PF4 cause some processing to occur. In the routine `show_my_window` when `mywindow` is displayed, only the TRANSMIT key will cause some processing to occur. In any other case, the while loop will continue and `sm_input` will be called again.

Close a Window

During the run of any application, there is always a form displayed. When a new form is displayed, all existing screens are implicitly closed. Windows, however, need to be explicitly closed if the application is to retreat to an underlying screen. For this reason, the `show_my_window` routine closes the window when it returns to the main program with a call to `sm_close_window`.

Reset the Terminal

Before the application terminates, it calls `sm_resetcrt` to reset terminal characteristics to a state expected by the operating system.

3. The developer may change the way messages are acknowledged with the library routine `sm_option` or via the setup file.

1.2
AUTHORING EXECUTABLE

The Authoring Executable must use the **JAM** Executive, and may have developer–written hook functions linked in. The main routine for the Authoring Executable is provided in source form in a file called `jxmain.c`. You may want to modify that file to change the default behavior of the authoring tool `jxform`, see section 1.3 below. It is strongly suggested that **JAM** developers read and understand this code, as it is instructive and may help with an understanding of the product.

Authoring executables must be linked with the **JAM** Authoring Library `jx`, the **JAM** Executive library `jm`, the Screen Manager library `sm`, and, in general, the standard math library. Since these executables are linked with the **JAM** Authoring Library `jx`, they may not be re–sold or distributed on machines for which there is no software license from JYACC. This restriction applies *only* to Authoring Executables, which are intended for application *development* only.

1.3
MODIFYING PROVIDED SOURCE CODE, JMAIN.C AND JXMAIN.C

The source files `jmain.c` and `jxmain.c` are very similar. They may be modified by developers who wish to change the default behavior of **JAM**. `jmain.c` is modified to change the way the application executable behaves, and `jxmain.c` is modified to change the way the authoring executable behaves.

Neither of the two main files should be used for the installation of hook functions. Hook functions are declared and installed in the file `funclist.c` as discussed below in section 2.1.2.

For the following discussion, it may be useful to obtain a listing of the source files from your system, namely `jmain.c` and `jxmain.c`.

Both provided main source files have four functions defined in them. The function `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 may be inserted into the `main` routine. Any code inserted before `initialize` will be executed before any **JAM** function has been executed. Code inserted after `initialize` but before `start_up` will be executed after **JAM** has allocated internal data structures and set the terminal characteristics, but

before there are any screens and before there is a local data block. Code inserted after `start_up` but before `clean_up` will be executed after JAM has exited the last screen, but before memory structures are de–allocated and the terminal is reset. Code called after `clean_up` will be executed after all JAM functions have been executed. If a finer granularity is needed, the functions `initialize`, `start_up`, and `clean_up` can themselves be edited. This should only be undertaken by developers with a very firm understanding of the product.

After the definition of the `main` function, there are a number of JAM sub–system macro definitions. They are all set to 0 by default. To turn on a sub–system, set the corresponding macro to 1. The sub–systems available are listed and described below:

- `SOFTKEYS`

  The JAM user interface supports soft keys. If this sub–system is enabled, JAM will use the soft key interface, displaying them on terminals that support them in hardware and simulating them on terminals that do not. For more information on soft keys, please see the *Author's Guide*. The use of soft keys will increase the application's memory requirements.

- `ALT_SCROLLING`

  If the application installs and uses one or more custom scroll driver as described in Chapter 10, this sub–system must be enabled. This will increase the application's memory requirements, unless the custom scroll driver saves memory that would otherwise be used for arrays by JAM.

- `MEMORY_SCREENS`

  If this sub–system is installed, any screens displayed by the JAM library itself will be linked into the application as data and maintained in memory. If this is not installed, the screens will be accessed in screen libraries on disk. Installing this sub–system will increase the application's memory requirements, but will speed up execution.

- `MEMORY_KEYSETS`

  If this sub–system is installed, the keysets used by the JAM library itself in providing a soft key interface will be linked into the application as data and maintained in memory. It only makes sense to install this sub–system if the `SOFTKEYS` sub–system is installed. Installing this sub–system will increase the application's memory requirements, but will speed up execution.

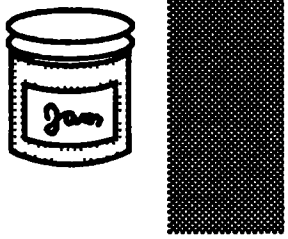
- `BLOCK_MODE`

  If the application installs and uses a block mode terminal driver as described in Chapter 11, this sub–system must be installed. It will increase the application's memory requirements.

- `JTERM_COMPRESSION`

  Installing this sub–system will increase the communication efficiency and

execution speed for applications when they are accessed by the terminal emulator ***Jterm***. It will increase the application's memory requirements.



Chapter 2
Hook Functions

The primary coding task facing JAM programmers is writing hook functions. These functions, which are called by the JAM Executive and by the Screen Manager when certain well–defined events occur, are written in C[4].

Hook functions can be used for many purposes. The following example shows a default field function that places the name, occurrence number, and field number of the current field on the status line whenever the field is entered. The arguments are those that are passed, by default, to a hook function designated as a field function. (Please note that this same function is discussed in depth later in section 2.2.1 on page 19.)

```
/* Include Files */
#include "smdefs.h"     /* Screen Manager Header File */

int
dfield ( 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 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 )
```

4. Hook functions may also be written in other third–generation programming languages for which JYACC supports a language interface. In particular, Fortran, Cobol and PL/1 are available for JAM on some platforms.

```
        {
                /* Obtain the field name */
                /* Format the status line */
                if ( * ( f_name = sm_name ( f_number ) ) )
                        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 ) ;

        }

        /* Return code of zero means that everything is fine. */
        return ( 0 ) ;
}
```

This function is installed with the following code normally added to the provided source code in `funclist.c`. Since it is installed as a DFLT_FIELD_FUNC, it is executed every time any field is entered, exited, or validated.

```
        .
        .
        .
        extern int dfield ( ) ;
        struct fnc_data dfield_struct = { 0, dfield, 0, 0, 0, 0 } ;
        .
        .
        .
        void
        sm_do_uinstalls ( ) ;
        {
        .
        .
        .
                sm_install(DFLT_FIELD_FUNC,&dfield_struct,(int *)0);

        .
        .
        .
        }
```

In this chapter, we discuss how hook functions are written and installed. They must also be compiled and linked into the JAM Application (or Authoring) Executable: see the Installation Guide for details of that. We also discuss what JAM events have hooks accessible to developers and what arguments are passed to hook functions from any given

hook. Finally, we discuss in detail the various types of hook functions, showing examples of how they might be written, installed, and used.

2.1 PREPARATION AND INSTALLATION

Hook functions, once properly installed, are called at certain well–defined JAM events. These events are outlined below in section 2.1.1 and discussed in detail later in the chapter.

There are many events that have hooks accessible to developers. JAM passes different arguments to the various hook functions, and interprets the return codes differently for each one. It is important that hook functions process the arguments that are passed correctly, and that they return meaningful codes based on the events to which they are attached.[5]

Some hook functions are installed individually, and are called at runtime by JAM when a certain event type occurs. Other hook functions, namely those attached to control strings, screen entry/exit, group entry/exit/validation, or field entry/exit/validation are installed in tables known as *function lists*. Since such functions are referred to with a string in screen binaries, the JAM Application Executable uses the function list to resolve those strings to actual function pointers at runtime. Most hook functions are called by the Screen Manager. However, the hook functions invoked with control strings are called by the JAM Executive, and are therefore accessible to applications using a custom executive only through JPL.

2.1.1 Types of Hook Functions

There are twenty–two installable hook function types, six of which are for the function lists and sixteen of which are for individual functions. They are briefly outlined below, and discussed in detail later in the document:

- `FIELD_FUNC`

  This is a function list. The functions on this list may be designated in the Screen Editor to be called by the Screen Manager as field entry, exit or validation functions for specific fields. The JPL `atch` verb may also be used to access these functions.

5. For certain types of hooks, you can specify the arguments that are passed to specific functions. See section 2.3 on page 66 on prototyped functions.

- GROUP_FUNC

  This is a function list. The functions on this list may be designated in the Screen Editor to be called by the Screen Manager as group entry, exit or validation functions for specific groups (Radio Buttons and Checklists).

- SCREEN_FUNC

  This is a function list. The functions in this list may be designated in the Screen Editor to be called by the Screen Manager as screen entry or exit functions on particular screens.

- CONTROL_FUNC

  This is a function list. These functions may be entered and invoked from control strings. They are often associated with function keys and menus in the Screen Editor or with the `sm_putjctrl` library call. The JPL `call` verb can invoke the functions in this list.

- PROTO_FUNC

  This is a function list for prototyped functions. Field, screen, group, and control functions may all be placed on this list together if they are prototyped. The prototyping of function list functions is discussed in section 2.3.

- DFLT_FIELD_FUNC

  This is an individual function. Once installed, it is called on entry, exit and validation for all fields.

- DFLT_GROUP_FUNC

  Similar to the DFLT_FIELD_FUNC, this individual function is called on entry, exit, and validation for all groups.

- DFLT_SCREEN_FUNC

  Individual function called on entry and exit for all screens.

- ASYNC_FUNC

  Individual function called asynchronously when JAM is waiting for keyboard input. Often used to poll external systems for mail delivery or the availability of data over a communications line.

- KEYCHG_FUNC

  Individual function called whenever JAM reads a key from the keyboard. This allows for the application to intercept and process (and possibly translate) keystrokes at the logical key level.

- INSCRSR_FUNC

  Individual function called by JAM whenever the keyboard entry mode toggles between insert and overstrike mode. This allows an application to update the display, if desired, to provide an indication of the new mode. Often used if there is no ability to change cursor styles between insert and overstrike modes.

- CKDIGIT_FUNC
  Individual function called by JAM for check digit validation of numeric fields. Only necessary if the default check–digit algorithm provided with JAM is not sufficient.

- UINIT_FUNC
  Individual function called just before the Screen Manager and the physical display are initialized at the start of the application.

- URESET_FUNC
  Individual function called just after the Screen Manager and the physical display are closed and reset at the end of the application, even if the application aborts ungracefully.

- RECORD_FUNC
  Individual function used to record keystrokes so they can be played back for tutorials or for regression testing.

- PLAY_FUNC
  Individual function used to playback recorded keys.

- AVAIL_FUNC
  Individual function used in advanced record/playback algorithms.

- STAT_FUNC
  Individual function used to intercept JAM status line processing and alter or replace it.

- VPROC_FUNC
  Individual function used to intercept JAM video processing and to alter or replace it.

- SCROLL_FUNC
  Function list of alternate field scrolling methods discussed in section 10.1. The functions on this list may be designated in the Screen Editor to be called by the Screen Manager as the alternative scroll driver for specific fields.

- DFLT_SCROLL_FUNC
  This is an individual function. Once installed, it is called as the scroll driver for all fields that do not have SCROLL_FUNC functions specified in the Screen Binary.

- BLKDRVR_FUNC
  This is an individual function that acts as a block mode terminal driver. This is discussed in section 11.1.3.

2.1.2

Provided Source Code — `funclist.c`

The file `funclist.c` is provided in source form with JAM. There is detailed documentation in that file about function installation. Many users will find it expedient to directly edit that file, inserting declarations, definitions and code for hook function installation.

The provided `funclist.c` is broken into five sections and has copious comments. The first section contains the necessary header files supporting function installation.

In the second section of `funclist.c`, a number of functions are declared. The function `sm_do_uinstalls()` is declared globally. This is the function called from the main routines in `jmain.c` and `jxmain.c`, and it is defined toward the end of `funclist.c`. A number of static declarations, for example, dummy hook functions, comes next. Those dummy functions are defined at the end of `funclist.c`. Developers should add definitions for their own hook functions at this point.

In the third section of `funclist.c`, a number of data structures supporting installation of the dummy hook function lists are defined. These definitions may be augmented with or replaced by the developer's own data structures and function list elements.

In the fourth section of `funclist.c`, the function `sm_do_uinstalls()` is defined. This function is called, generally by the main routine, to install all the necessary hooks. Note the calls to the library function `sm_install`. These calls install the dummy function lists. Developers should add their own installation calls at this point.

In the fifth section of `funclist.c`, the dummy hook functions are defined. Developers may place their own hook functions here, or they may be external to `funclist.c` and found in separate source files.

2.1.3

Preparing Functions for Installation

Once hook functions are written, they must be included in some JAM data structures prior to installation. Many users will find it expedient to add definitions and declarations used to prepare functions for installation in the provided source file `funclist.c`. Individual hook functions are stored in a structure of type `fnc_data` in preparation for installation. Function lists are stored in arrays of `fnc_data` structures. The `fnc_data` structure is shown and described below:

```
struct fnc_data
{
        char *fnc_name ;        /* Function Name */
        int ( *fnc_addr )( ) ;  /* Function Address */
        char language ;         /* Function language */
        char intrn_use ;        /* Installation Parameter */
        char appl_use ;         /* Byte for Developer */
        char reserved ;         /* Reserved by JYACC */
}
```

- `fnc_name`

  A character string naming the hook function. It need only be named here if it is a function in a function list, since JAM needs to resolve the text string name of a function attached to a screen, group, field, or control string to a function address. If the function is not in a function list, this pointer should be 0.

- `fnc_addr`

  The address of the routine, namely the C identifier used for the function in your source code.

- `language`

  Code for the language the function is written in. Refer to the header file `smidenty.h`. Use 0 for C.

- `intrn_use`

  Installation parameter.

- `appl_use`

  Reserved for application use.

- `reserved`

  Reserved for future releases. Set this to 0.

Functions to be individually installed (i.e. not in functions lists) are also placed into a structure of type `fnc_data`. To create a `fnc_data` structure identified as `my_keychg_struct` with an individual function identified as `my_func`, use the following definition, generally placing it in the file `funclist.c`:

```
#include "smdefs.h"
extern int my_keychg_struct ( ) ;
struct fnc_data my_keychg_struct = { 0, my_func, 0, 0, 0, 0 };
```

Function lists are implemented as arrays of `fnc_data` structures. To create a function list called `my_field_list`, with the functions `my_func1`, `my_func2`, and

my_func3, use the following definition, generally placing it in funclist.c[6]:

```
#include "smdefs.h"
extern int my_func1 ( ), my_func2 ( ), my_func3 ( ) ;
struct fnc_data my_field_list[] = {
                { "do_entry", my_func1, 0, 0, 0, 0 },
                { "do_validation", my_func2, 0, 0, 0, 0 },
                { "do_exit", my_func3, 0, 0, 0, 0 },
                { "fld_exit", my_func3, 0, 0, 0, 0 }
} ;
int my_field_size = sizeof ( my_field_list ) /
                        sizeof ( struct fnc_data ) ;
```

The integer my_field_size is defined and initialized because the address of an integer with the number of functions in a function list must be passed to the installation routine when the function list is installed.

2.1.4

Installing Functions

Hook functions are installed with the library routine sm_install. Most developers will find it expedient to add their installation code to the function sm_do_uninstalls provided in source form in the file funclist.c. In some cases, developers may want to call sm_install from other points in their applications.

As is documented formally in the Programmer's Reference, sm_install is passed three arguments. The first argument is the type of function or function list to be installed. The second argument is the address of the fnc_data structure or array of structures to install, and the final argument is a pointer to an integer. The third argument should be set to (int *)0 when an individual function is installed. When a function list is installed, the third argument is the address of an integer with the number of entries in the list.

To install the function my_func as the application key change function, after the declarations and definitions shown above in section 2.1.3, add the following call to sm_do_uinstalls:

```
sm_install ( KEYCHG_FUNC, &my_keychg_struct, (int *)0 ) ;
```

To install the functions my_func1, my_func2, and my_func3 into the application field function list after the declarations and definitions shown above in section 2.1.3, add the following call to sm_do_uinstalls:

6. Note that in this example, either the string "do_exit" or the string "fld_exit", when appropriately designated as a field function in the Screen Editor, will cause the function my_func3 to be executed at runtime. Possible uses for this technique of giving the same function different names include mapping functions yet to be written to a stub routine, and using the same function to perform slightly different tasks (with the name as an implied parameter).

```
sm_install ( FIELD_FUNC, my_field_list, &my_field_size ) ;
```

Note that the final argument to `sm_install` is the *address* of an integer. This is so that, in the case of function lists, the size of the new list can be returned to the calling application.

For information on linking a function into **JAM** and creating new executables, see the Installation Guide.

2.2
WRITING HOOK FUNCTIONS

Arguments passed to hook functions and return values received from hook functions vary from hook to hook. In this section, we discuss the various **JAM** hooks in detail.

2.2.1
Field Functions

The Screen Manager calls field functions, if specified and installed, on field entry, field exit, and field validation. Calls to field entry and field exit functions are guaranteed to be paired for any given field.

A single default field function may be installed. It will be invoked on entry, exit, and validation for every field. Field functions specified as entry, exit, or validation functions for a given field via the Screen Editor must be installed into the field or prototype function list. The default field function is installed separately from the list, and need not appear in the list. JPL procedures may also be directly specified as field functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl fieldfunc`. Such JPL procedures should not be in the function list.

The default field function is installed as `DFLT_FIELD_FUNC`. Field functions to be added to the field function list are installed as `FIELD_FUNC`.

Field Function Invocation

Field functions are called for field entry whenever the cursor enters a field, including when the field containing the cursor is activated by virtue of an overlying window being closed. Field functions are called for field exit whenever the cursor leaves a field, including when the field is exited because a window is popped up over the existing screen. Field functions are called for validation whenever the field is validated. This occurs at the following times:

- As part of field validation, when you exit the field or scroll to the next occurrence by filling it or by hitting TAB or RETURN key. The BACKTAB and arrow keys do not normally cause validation. Field functions are called for validation only after the field's contents pass all other validations for the field.
- As part of screen validation when the XMIT key is struck.
- When the application code calls library routines for field validation.

Field functions on the `FIELD_FUNC` function list may also be invoked from JPL with the `atch` verb.

For fields that are members of menus, radio buttons, or checklists, the validation function is not called as part of validation. The validation function for such fields is called instead when that field is selected. For checklist fields, the field validation function is also called when the field is deselected.

Field functions specified for field entry via the Screen Editor are invoked after any installed default field function. Field functions specified for field exit or validation via the Screen Editor are called before any installed default field function.

Field Function Arguments

All field functions receive four arguments:

1. The field number as an integer.
2. A pointer to a null terminated character string containing a copy of the field's contents.
3. The occurrence number of the data as an integer.
4. An integer bitmask containing contextual information about the validation state of the field and the circumstances under which the function was called.

The contextual information in the last parameter includes the following bit masks[7]:

- `VALIDED`

  If this is set (i.e. `if (param4 & VALIDED)`), the field has passed all its validations and has not been modified since.

- `MDT`

  If this is set (i.e. `if (param4 & MDT)`), the field data has been

7. The example field function below contains a procedure called `bitmask` that is useful for checking whether a particular flag (bit location in a binary value) is set. Source code for this procedure can also be found in the `sample` application provided with JAM.

changed either from the keyboard or from the application code since the current screen was opened[8]. JAM never clears this bit. The application code may clear it directly with the `sm_bitop` library routine.

- `K_ENTRY`

  If set (i.e. `if (param4 & K_ENTRY)`), the field function was called on field entry.

- `K_EXIT`

  If set (i.e. `if (param4 & K_EXIT)`), the field function was called on field exit[9].

- `K_EXPOSE`

  If set (i.e. `if (param4 & K_EXPOSE)`), the field function was called because a window overlying the screen on which the field resides was opened or closed[10].

- `K_KEYS`

  Mask for the bits indicating which keystroke or event caused the field to be entered, exited, or validated. The intersection of this mask and the fourth parameter to the field function should be tested for equality against one of the six remaining values below:

- `K_NORMAL`

  If set (i.e. `if ((param4 & K_KEYS) == 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".

- `K_BACKTAB`

  If set (i.e. `if ((param4 & K_KEYS) == K_BACKTAB)`), the BACKTAB key caused the cursor to enter or exit the field in question.

- `K_ARROW`

  If set (i.e. `if ((param4 & K_KEYS) == K_ARROW)`), an arrow key caused the cursor to enter or exit the field in question.

- `K_SVAL`

  If set (i.e. `if ((param4 & K_KEYS) == K_SVAL)`), the field is being validated as part of screen validation.

8. Note that when the screen is being opened, when the screen entry function modifies data in a field the MDT bit is not set. However, when the screen is exposed by virtue of an overlaid window being closed, modification of field data in the screen entry function causes the MDT bit to be set.

9. Note that if neither `K_ENTRY` nor `K_EXIT` are set, the field is being validated.

10. This means that if both `K_ENTRY` and `K_EXPOSE` are set, the field is being exposed. If `K_EXIT` and `K_EXPOSE` are set, the field is being hidden.

- K_USER

  If set (i.e. `if ((param4 & K_KEYS) == K_USER)`), the field is being validated directly from the application with the `sm_fval` library routine.

- K_OTHER

  If set (i.e. `if ( (param4 & K_KEYS) == K_OTHER)`), some key other than backtab, arrow or those mentioned as "normal" caused the cursor to enter or exit the field in question.

Field functions are called for validation regardless of whether the field was previously validated. They may test the VALIDED and MDT bits to avoid redundant processing.

Field Function Return Codes

Field functions called on entry or exit should return 0 if they do not reposition the cursor. 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 is not repositioned to the field in question. Any other non–zero return value causes the cursor to be repositioned to the field. This repositioning is useful when an entire screen is undergoing validation, since the field that fails validation may not be the field where the cursor lies.[11]

Example Field Function List

The following field functions, intended to be installed in a list and consequently accessible to field function designations in screens, do field initialization and validation based on external criteria:

```
/*
* Two field functions for inclusion on the field function list
* are defined here. The first one, fentry(), initializes the
* value in a field provided that it has not been 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 will clearly be application specific.
*
* The field function list is defined and declared as follows:
*
*       )
*
*       extern int fvalid ( ) ;
```

11. In many cases, it is better for the field validation function itself to reposition the cursor before displaying an error message, otherwise the error message might be misleading.

```
*       extern int fentry ( ) ;
*
*       struct fnc_data ffuncs[] =
*       {
*               { "fentry", fentry, 0, 0, 0, 0 },
*               { "fvalid", fvalid, 0, 0, 0, 0 }
*       } ;
*
*       int fcount = sizeof ( ffuncs ) /
*                       sizeof ( struct fnc_data ) ;
*
* The field function list is then augmented with the following
* call:
*
*       sm_install ( FIELD_FUNC, ffuncs, &fcount ) ;
*
*/

/* 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_err_reset ( msg_buf ) ;

                /* Return code of 1 indicates validation fail */
                return ( 1 ) ;
        }

        return ( 0 ) ;
}
```

Example Default Field Function

The following field function, intended to be installed as the default field function for the entire application, maintains a status line of field identification information:

```
/*
* This function is intended to be installed as the default field
* function in a JAM application. It is called on the entry,
* exit, and validation for all fields
*
* The following declarations/definitions would commonly be
* included in the main function source module or in the source
* file funclist.c:
*
*       extern int dfield ( ) ;
*       struct fnc_data dfield_struct =
*               { 0, dfield, 0, 0, 0, 0 } ;
*
* This function is installed with the following line of code,
* often found in the main function or in the function
* sm_do_uinstalls() in the file funclist.c:
*
*       sm_install ( DFLT_FIELD_FUNC, &dfield_struct,
*                       (int *) 0 ) ;
*
* The function causes entry into a field to place identifying
* information for that field on the status line. When fields
* that are members of groups (radio buttons or checklists)
* are selected, a message will be displayed on the status line
* about that event.
```

```
*
* Identifying information on fields will be the name, if it
* exists, the number, and the occurrence number. Selection
* events in groups will show the text of the selected field,
* the group name, and the group occurrence.
*/

/* Include Files */
#include "smdefs.h"     /* Screen Manager Header File */

int
dfield ( 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 ) ;
}
```

2.2.2
Screen Functions

The Screen Manager calls screen functions, if specified and installed, on entry and exit of screens. Calls to screen entry and screen exit functions are guaranteed to be paired for each screen.

A single default screen function may be installed. It will be invoked on entry and exit for every screen. Screen functions specified as entry or exit functions for a screen via the Screen Editor must be installed into the screen or prototype function list. The default screen function is installed separately from the list, and need not appear in the list. JPL procedures may also be directly specified as screen functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl screenfunc`. Those procedures should not be in the function list.

The default screen function is installed as DFLT_SCREEN_FUNC. Screen functions to be placed on the screen function list are installed as SCREEN_FUNC.

Because of the way LDB processing and form stack handling is done, it is neither recommended nor supported to call any form or window display library routines from

screen entry or exit functions. If it is necessary to display windows at screen entry, the library routine `sm_ungetkey` can be invoked, passing as the argument a function key with a control string that brings up a window.

Screen Function Invocation

Screen functions are called for screen entry whenever a screen is opened. Screen functions are called for screen exit whenever a screen is closed. Optionally, screen functions may also be called for entry when a screen is exposed by virtue of a window overlaying it being closed or deselected, and called for exit when a window is popped up or selected over the screen in question.[12] This is not the default behavior because it would introduce incompatibilities with earlier releases of JAM.

If you are not concerned with compatibility with earlier releases, it is strongly suggested that you enable the calling of screen functions when screens are exposed or hidden. This may be done either by setting the `EXPHIDE_OPTION` to `ON_EXPHIDE` in the setup file (refer to the *Configuration Guide*) or making the following library function call near the beginning of your application:

```
sm_option(EXPHIDE_OPTION, ON_EXPHIDE)
```

Screen functions specified for screen entry via the Screen Editor are invoked after any installed default screen function. Screen functions specified for screen exit via the Screen Editor are called before any installed default screen function.

Screen Function Arguments

All screen functions receive two arguments:

1. A pointer to the null terminated character string containing the screen name.
2. An integer bitmask containing contextual information about the circumstances under which the function was called.

The contextual information in the second parameter includes the following bit masks:

- `K_ENTRY`

  If this is set (i.e. `if (param2 & K_ENTRY)`), the function was called on screen entry.

- `K_EXIT`

  If this is set (i.e. `if (param2 & K_EXIT)`), the function was called on screen exit.

12. Not that expose/hide processing is not performed when error windows are opened or closed

- K_EXPOSE

  If this is set (i.e. `if (param2 & K_EXPOSE)`), the function was called because the screen was selected or deselected, or because a window was popped over the screen or a window that used to be overlaid on the screen was closed[13].

- K_KEYS

  Mask for the bits indicating which event caused the screen to be exited. The intersection of this mask and the second parameter to the screen function should be tested for equality against one of the two remaining values below:

- K_NORMAL

  If set (i.e. `if ((param2 & K_KEYS) == K_NORMAL)`), a "normal" call to `sm_close_window` caused the screen to close.

- K_OTHER

  If set (i.e. `if ((param2 & K_KEYS) == K_OTHER)`), the screen is being closed because another form is being displayed or because `sm_resetcrt` is called.

Screen Function Return Codes

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 again.

Example Default Screen Function

The following screen function, intended to be installed as the default screen function for the entire application, maintains information about how long screens are open, and the aggregate amount of time they are active. Note the use of the P_USER pointer, a general purpose pointer you can manipulate that JAM will associate with an open screen.

```
/*
* This function is designed to keep 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.
```

13. If both K_ENTRY and K_EXPOSE are set, the screen is being uncovered and activated by virtue of an overlaid window being closed. If both K_EXIT and K_EXPOSE are set, the screen is being covered and deactivated by virtue of a window being popped up over it.

```
*
* The following declarations/definitions would typically be
* found in the main function source module or in the source
* file funclist.c:
*
*       extern int dscreen ( ) ;
*       struct fnc_data dscreen_struct =
*               { 0, dscreen, 0, 0, 0, 0 } ;
*
* The following line of code, typically found in the main
* function or in the function sm_do_uinstalls(), installs
* this function as the default screen function:
*
*       sm_install ( DFLT_SCREEN_FUNC, &dscreen_struct,
*                       (int *)0 ) ;
*
* For this function to operate correctly, the Screen Manager
* option to call hook functions on EXPOSE and HIDE must also
* be in place. That is set, generally in the main function,
* with the following call:
*
*       sm_option ( EXPHIDE_OPTION, ON_EXPHIDE ) ;
*
* To store aggregate times, this function utilizes the JAM 5.0
* feature that allows a user to associate a pointer with a
* screen.
*
* 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
dscreen ( 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 ) ;
}
```

2.2.3

Control Functions

Control functions are called by the JAM Executive in the processing of control strings and by JPL routines that call C functions. The JAM Executive calls control functions, if specified and installed, when control strings that start with a caret (^) are executed. JPL procedures may also execute control functions by using the `call` verb.

There is no default control function. Control functions must be installed in a function list. JPL procedures may be directly specified as control functions by preceding the name of the procedure in a control string with the string "`jpl `". JPL procedures specified in control strings in this manner need not be placed in the function list.

A number of control functions of general use are built in to JAM and pre-installed in the control function list. These built-ins can be used by any JAM application. They are listed in Chapter 4.

Control functions to be placed on the control function list are installed as `CONTROL_FUNC`.

Control Function Invocation

Control functions are called by the JAM Executive when a control string starting with a caret is processed. Such control strings are often attached, via the Screen Editor, to function keys or to menu selections in control fields. In addition, the JPL verb `call` can be used to invoke functions installed on the control function list[14].

14. The JPL call verb does not execute control strings. It does look for functions to call on the control function list.

Control Function Arguments

Control functions receive a single argument, namely a pointer to a copy of the control string that invoked them without the leading caret. It is only the first word on the control string that identifies the function, the rest of the string may contain arbitrary data that can be parsed and used as arguments.

Control Function Return Codes

Control functions may return any integer. The return value from a control function may be used for conditional control branching in target lists (see the *Author's Guide*). If the function returns a function key that is not a value in the target list, JAM executes the control string associated with the key.

Example Control Function List

The following example shows two closely related functions that would be appropriate for inclusion on a control function list[15].

```
/*
* Two functions intended as installed CONTROL_FUNC functions
* follow below. The function mark_low() will cause all fields on
* the current screen which have numeric values less than zero to
* be marked with an attribute change. The function mark_high()
* will cause all fields on the current screen which have numeric
* values higher than 1000 to be marked.
*
* The functions are installed so they can be called from JAM
* control strings. The following definitions and declarations to
* support the installation are generally found in the main
* function source module or in the source file funclist.c:
*
*       extern int mark_low ( ) ;
*       extern int mark_high ( ) ;
*       struct fnc_data mark_funcs[] = {
*               { "mark_low", mark_low, 0, 0, 0, 0 },
*               { "mark_high", mark_high, 0, 0, 0, 0 }
*       };
*       int mark_count = sizeof ( mark_funcs ) /
*                        sizeof ( struct fnc_data ) ;
*
* The installation of this function list is completed with the
* following line of code, typically found in the main function
* or in the function sm_do_uinstalls() found in funclist.c:
*
```

15. The same functionality shown here is shown in a better example in the section on prototyped functions. See section 2.3.

```
*       sm_install ( CONTROL_FUNC, mark_funcs, &mark_count ) ;
*
* 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
*
* As an alternative, you could prototype a single 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
#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 ) ;
}
```

Advanced Control Function Example

The following example shows how a number of entries in a control function list might map to the same function which uses the identifying string as an implied first argument. Significant argument processing is done in this example:

```
/*
* The following function will create 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.
*
* This function is intended to be installed in a function list
* for JAM control functions, namely the CONTROL_FUNC list. When
* a control string calling this function is invoked, the entire
```

```
* control string is passed as an argument to this function. The
* name with which the function is invoked is an implied argument,
* and specifies which report should be generated: field, screen,
* window stack, or display. The remainder of the control string
* specifies what * should be done with the report output. This
* could 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 either in
```

```
* funclist.c or in the main source module:
*
*       extern int report ( ) ;
*       struct fnc_data report_funcs[] = {
*               { "rep_field", report, 0, 0, 0, 0 },
*               { "rep_screen", report, 0, 0, 0, 0 },
*               { "rep_wstack", report, 0, 0, 0, 0 },
*               { "rep_term", report, 0, 0, 0, 0 }
*       } ;
*       int report_count = sizeof ( report_funcs ) /
*                       sizeof ( struct fnc_data ) ;
*
* Notice that the function list is constructed with four function
* entries with different names all of which refer to the same
* function pointer. Since, in the case of CONTROL_FUNC
* functions, the entire control string is passed to the called
* function in a string, the name with which the function was
* invoked can (and in this case does) serve as an implied
* argument.
*
* The actual installation of the function is done with the
* following library routine call, generally found either in
* the main routine or in sm_do_uinstalls(), defined in
* funclist.c:
*
*       sm_install ( CONTROL_FUNC, report_funcs,
*                       &report_count ) ;
*
*/

/* 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 the report output designator to the 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 -- flushed with
                    call to err_reset */
            sm_mwindow ( ptr, -1, -1 ) ;
            sm_err_reset ( "Report Done..Hit Space to "
                "continue" ) ;

            /* Close message window */
            sm_close_window ( ) ;

            /* Free up the malloced buffer. */
            free ( ptr ) ;
            break ;
```

```
        default :
                /* File appended, just close it. */
                fclose ( fp ) ;
                sm_err_reset ( "File appended" ) ;
                break ;
        }
        return ( 0 ) ;
}
```

2.2.4

Key Change Functions

The key change function is called by the Screen Manager as keys are read from the keyboard from within the library routine `sm_getkey`, which is called in the input processing for all keys by JAM. Only one individual keychange function may be installed at a time.

Keys placed on the queue with the library routine `sm_ungetkey` or with the built–in control function `^jm-keys` are not processed by the installed key change function.

The key change function is installed as `KEYCHG_FUNC`.

Key Change Function Invocation

The key change function is called exactly once for every key read in from the keyboard or supplied by the playback hook function described in section 2.2.10.

Key Change Function Arguments

The key change function is passed a single integer argument, namely the JAM logical key that was read from the keyboard or received from the playback hook function.

Key Change Function Return Codes

The key change function returns the key to be substituted for the one passed as an argument. Any key returned to `sm_getkey` will be returned by `sm_getkey` to its caller. However, if the key change function returns 0, `sm_getkey` will get the next key from the keyboard[16].

Example Key Change Function

The following key change function intercepts times when the user enters an exclamation point or strikes the EXIT key. Another example key change function is shown as part of the group function example on page 47.

16. See the library routine `sm_keyoption` for a different method of changing the function of a logical key.

```
/*
* This function, installed as an application keychange function,
* will cause sm_getkey to intercept two keys, the exclamation
* point and the logical EXIT key. When the user types an
* exclamation point, this function will ask 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 does escape to the shell, or if the user
* does not want to EXIT, this function gobbles up the keystroke.
* If the user does not want the shell, or really does want to
* EXIT, the keystroke is passed back to sm_getkey.
*
* Note also the preprocessor directives about whether or not the
* JAM executive is in use. If the JAM Executive is in use, we do
* not bother querying about the EXIT if there are control strings
* associated with EXIT. In addition, we can use the standard JAM
* operating system escape.
*
* This function is installed by including the following
* definitions and declarations, generally in the main function
* source file or in the source file funclist.c:
*
*       extern int keychg ( ) ;
*       struct fnc_data keychg_struct =
*               { 0, keychg, 0, 0, 0, 0 } ;
*
* and with the following line of code, generally near the top of
* the main function or in the funclist.c function
* sm_do_uinstalls():
*
*       sm_install ( KEYCHG_FUNC, &keychg_struct, (int *) 0 ) ;
*
*/

/* Include Files */
#include "smdefs.h"     /* Screen Manager Header File */
#include "smkeys.h"     /* Screen Manager Logical Keys */

#define EXIT_CONFIRM "Are you sure you want to EXIT? (y/n)"
#define SHELL_CONFIRM "Are you sure you want to go to OS? (y/n)"

int
keychg ( the_key )
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, then 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 ) ;
}
```

2.2.5

Group Functions

The Screen Manager calls group functions, if specified and installed, on entry, exit, and validation of radio buttons and checklists. Calls to group entry and group exit functions are guaranteed to be paired for each group.

A single default group function may be installed. It will be invoked on entry, exit, and validation for every group. Group functions specified as entry, exit, or validation functions for a given group in the Screen Editor must be installed into the group or prototype function list. The default group function is installed separately from the list, and need not appear in the list. JPL procedures may also be directly specified as group functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl groupfunc`. Those procedures should not be in the function list.

The default group function is installed as `DFLT_GROUP_FUNC`. Group functions to be placed on the group function list are installed as `GROUP_FUNC`.

Please note that field validation functions for fields that are members of groups or menus are called at selection and, in the case of checklists, deselection as discussed above in section 2.2.1 on page 19.

Group Function Invocation

Group functions are called for group entry whenever the cursor enters a group, including the times when the group containing the cursor is activated by virtue of an overlying window being closed. Group functions are called for group exit whenever the cursor leaves a group, including the times when the group is left because a window is popped up over the existing screen. Group functions are called for validation whenever the group is validated. This occurs at any of the following times:

- As part of group validation, when you exit the group by hitting TAB or making a selection from an autotab group. The BACKTAB and arrow keys do not normally cause validation.
- As part of screen validation when the XMIT key is struck.
- When the application code calls library routines for group validation.

Group functions specified for group entry via the Screen Editor are invoked after any installed default group function. Group functions specified for group exit or validation via the Screen Editor are called before any installed default group function.

Group Function Arguments

All group functions receive two arguments:

1. A pointer to the null terminated character string containing the group name.
2. An integer bitmask containing contextual information about the validation state of the group and the circumstances under which the function was called.

The bits that make up the bitmask for group functions are identical to those that are passed in the fourth argument to field functions, and are outlined in section 2.2.1 on page 20.

Group functions are called for validation regardless of whether the group was previously validated. They may test the VALIDED and MDT bits to avoid redundant processing.

Group Function Return Codes

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. Any non–zero return code should indicate that the group does not pass validation. If the returned value is 1, the cursor is not repositioned to the offending group. Any other non–zero return value causes the cursor to be repositioned to the group that failed the validation.

Example Default Group Function

The following group function, intended to be installed as the default group function for the entire application, installs a keychange function while the cursor is in groups that use check boxes:

```
/*
* Two functions are defined in this module. The first, dgroup(),
* is intended to be installed as a default group function, to be
* called when any group is entered, exited, or validated if a
* specific function is not specified for the group. The second,
```

```
* keychg(), is a keychange function to be installed on group
* entry and de-installed on group exit by dgroup().
*
* Note that pre-existing keychange functions are properly stacked
* by dgroup(). keychg() also chains existing keychange functions
* along, but it is assumed that they are written in C.
* Pre-existing keychange functions in some other supported
* third generation language may not be properly chained by this
* function.
*
* These functions enable selection of group fields by pressing
* the "X" key. They are designed for groups that have checkboxes,
* since it sometimes makes more sense to use the "X" key for
* selection than the NL key.
*
* The function would be installed by placing the following
* definitions and declarations in the main function source module
* or in the source file funclist.c:
*
*       extern int dgroup ( ) ;
*       struct fnc_data dgroup_struct =
*               { 0, dgroup, 0, 0, 0, 0 } ;
*
* and the following line of code in the main function itself, or
* in the function sm_do_uinstalls() in the file funclist.c:
*
*       sm_install ( DFLT_GROUP_FUNC, &dgroup_struct,
*                       (int *) 0 ) ;
*
*/

/* 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
dgroup ( name, context )
char *name ;            /* Group Name */
int context ;           /* Context bits */
{
        /* If the group does not have check boxes,
                we don't want this. */
        if ( ! sm_n_edit_ptr ( name, CKBOX ) )
                return ( 0 ) ;

        /* 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 ) ;
}
```

2.2.6

Asynchronous Functions

The installed asynchronous function is called periodically by the Screen Manager while the keyboard input routine waits for user input. It can be used to poll or otherwise manipulate communications resources, or to update the display on the screen. Only one asynchronous function may be installed at a time.

The asynchronous function is installed individually as ASYNC_FUNC.

Asynchronous Function Invocation

The asynchronous function is called from the very lowest level of JAM keyboard input. When the asynchronous function is installed, the device driver clock on the terminal input device is set to time out on its character read operation, and if a character is not read in that time interval the asynchronous function is invoked before another character read operation is attempted. The time out interval is specified at installation in the `intrn_use` field of the `fnc_data` structure. That time interval is measured in tenths of seconds. The maximum interval is 255 (25.5 seconds).

Asynchronous Function Arguments

The asynchronous function is passed no arguments.

Asynchronous Function Return Codes

The asynchronous function should generally return 0. If it returns –1, it will not be called again until at least one additional character has been read from the keyboard. The asynchronous function may return a key, which will be returned to `sm_getkey` and on to the application. If that key is a JAM logical key, no further translation will be done. If the asynchronous function returns a data character, JAM will interpret it as a physical keyboard stroke.

Example Asynchronous Function

The following example shows an asynchronous function that maintains a display of the current time on the status line. It should be installed to be called once per second.

```
/*
* This function, when run from a JAM application, will output the
* current time at the end of the status line.
*
* It can be installed with the following declarations and
* definitions, generally found in the main function source file
* or in the file funclist.c:
*
*       extern int async ( ) ;
*       struct fnc_data async_struct =
*               { 0, async, 0, 10, 0, 0 } ;
*
* and the following line of program code, possibly in the main
* function, or in the function sm_do_uinstalls() in the file
* funclist.c:
*
*       sm_install ( ASYNC_FUNC, &async_struct, (int *)0 ) ;
*
* For the asynchronous function, the fourth element in the
* fnc_data structure is the measure in tenths of seconds between
* calls. In the example above, the asynchronous function is
* called once every second that the keyboard is idle while being
* read.
*
* Note that this function assumes that the cursor position
* display is not being used.
*/

/* Include Files */
#include "smdefs.h"     /* Screen Manager Header File */
```

```
#include <time.h>       /* Needed for time() calls */

/* Buffer Sizes */
#define STAT_LINE_LEN 80
#define TIME_LEN 8

int
async ( )
{
        struct tm *tm_struct ;          /* System time */
        char tm_buf[ TIME_LEN + 1 ] ;   /* Time as string */
        time_t tm_t ;                   /* Result of time() */

        /*
        * First get the current time. Note that time() is an
        * ANSI standard function whose return is implementation
        * dependent.
        *
        * The localtime() call is part of the standard C library.
        */
        tm_t = time ( (time_t *)0 ) ;
        tm_struct = localtime ( &tm_t ) ;

        /* Format a character string with the current time */
        sprintf ( tm_buf, "%02d:%02d:%02d", tm_struct->tm_hour,
            tm_struct->tm_min, tm_struct->tm_sec ) ;

        /* Place that character string at end of status line. */
        sm_msg ( STAT_LINE_LEN - TIME_LEN - 1,
            TIME_LEN, tm_buf) ;

        /*
        * Returning 0 means that this function will continue
        * to be called every second.
        */
        return ( 0 ) ;
}
```

2.2.7

Insert Toggle Functions

The Screen Manager calls the Insert Toggle Function when switching between insert and overstrike mode for data entry. Generally this hook function is used to update some aspect of the display informing the user of the current mode.

The insert toggle function is installed individually as INSCRSR_FUNC. JAM automatically installs an insert toggle function that changes the cursor style when the mode is changed. If an application installs its own insert toggle function, the JAM function will be deinstalled, and the new insert toggle function may want to call the function directly.

Insert Toggle Function Invocation

The function is invoked by JAM whenever the data entry mode shifts from insert to overstrike mode or from overstrike to insert mode. Most often, this occurs when the end–user strikes the INSERT key.

Insert Toggle Function Arguments

One integer argument is passed to the insert toggle function. It specifies the mode. If its value is 1, JAM is entering insert mode. If it is 0, JAM is entering overstrike mode.

Insert Toggle Function Return Codes

The insert toggle function should return 0.

Example Insert Toggle Function

The following example shows a function that displays the current mode at the end of the status line:

```
/*
* This function is designed to be installed as the INSCRSR_FUNC
* hook function, called whenever JAM moves from insert to
* overstrike mode or vice versa. It places the letters INS
* on the status line in insert mode, and OVR on the
* status line for overstrike mode.
*
* The following declarations/definitions are generally found in
* the main function source file or the source file funclist.c
* to support the installation of this function:
*
*       extern int inscrsr ( ) ;
*       struct fnc_data inscrsr_struct =
*               { 0, inscrsr, 0, 0, 0, 0 } ;
*
* The following lines of code would typically be included in the
* main function or in the funclist.c function sm_do_uinstalls()
* to install this function:
*
*       sm_install ( INSCRSR_FUNC, &insrcrsr_struct,
*                       (int *)0 ) ;
*
* This routine assumes that cursor position display is not in
* use. You may also need a STAT_FUNC function for this, as JAM
* will overwrite the status line with messages, thus destroying
* the INS/OVR message.
*
```

```
* Note that we do not write in the last column of the status
* line, since JAM will not permit the writing to the last
* position of a screen if it would cause automatic hardware
* scrolling.
*/

/* Include Files */
#include "smdefs.h"     /* Screen Manager Header File */

/* Buffer Sizes */
#define STAT_LINE_LEN 80

int
inscrsr ( entering_insert_mode )
int entering_insert_mode ;      /* Non-zero if about to enter
                                        insert mode, zero
                                        if about to enter
                                        overstrike mode. */
{
        if ( entering_insert_mode )
                sm_msg ( STAT_LINE_LEN - 4, 3, "INS" ) ;
        else
                sm_msg ( STAT_LINE_LEN - 4, 3, "OVR" ) ;
        return ( 0 ) ;
}
```

2.2.8

Check Digit Functions

The Screen Manager calls the check digit function for any field that is marked for check digit in the Screen Editor. It may be used to implement any desired check-digit algorithm. If there is no check digit function installed in the application, JAM uses the default library function `sm_ckdigit` which is distributed in source form with JAM. If the default algorithm is not satisfactory and your linker permits overriding library functions, you may merely modify the provided source and link it to your application without installing it. If your linker does not permit the overriding of library functions, you will need to install a new check digit function with `sm_install`.

The check digit function is installed individually as `CKDIGIT_FUNC`.

Check Digit Function Invocation

The check digit function is called by JAM during validation of fields marked for check digit.

Check Digit Function Arguments

The check digit function is passed the following arguments:

1. The integer number of the field undergoing validation.
2. A character pointer to a buffer containing the field contents in a null–terminated string.
3. The integer occurrence number for the data undergoing validation.
4. The integer modulus as specified in the Screen Editor.
5. The integer minimum number of digits as specified in the Screen Editor.

Check Digit Function Return Codes

The check digit function should return 0 if the field passes the check digit validation. If a non–zero value is returned, the cursor is positioned to the offending field and the field is not marked as validated. It is assumed that the check digit function display its own error messages.

Please see the provided source code for `sm_ckdigit` as an example.

2.2.9

Initialization and Reset Functions

The initialization and reset functions are called by the Screen Manager on display setup and display reset respectively. The initialization function can be used to set the terminal type and the reset function can be used to handle any cleanup that the application needs to do whether it is terminated gracefully or not.

Initialization and reset functions are installed individually as `UINIT_FUNC` and `URESET_FUNC` respectively.

Initialization and Reset Function Invocation

The initialization function is called from the library routine `sm_initcrt`. When it is called, **JAM** has not yet allocated its required memory structures, and the physical display characteristics are still untouched by **JAM**. In general, it is suggested that hook functions be installed after initialization with `sm_initcrt`, but clearly this is an exception. The initialization function must be installed before `sm_initcrt` is called. Consequently, the installation code for this one hook function may not be placed in the `funclist.c` routine `sm_do_uinstalls`.

The reset function is called from the library routine `sm_resetcrt` after **JAM** has released its memory and reset the physical display characteristics. Since the **JAM** abort

routine sm_cancel calls sm_resetcrt before the application terminates, the reset function is generally called at application exit whether the exit is graceful or not[17].

Initialization and Reset Function Arguments

The initialization function is passed a single argument, namely a pointer to a 30 byte character buffer into which it may place the null–terminated string mnemonic identifying the terminal type in use. This is primarily of use on operating systems without an environment. This function can be used to obtain the terminal type in some system–specific way.

The reset function is passed no arguments.

Initialization and Reset Function Return Codes

Both the initialization and reset hook functions should return 0.

Example Initialization and Reset Functions

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 intended to
* be installed as the initialization and reset functions
* respectively. uinit() is used to initialize the global
* 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
* will cause 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.
*
* These functions are properly installed with the following
* definitions and declarations, generally found in the main
* function source module or in the source file funclist.c:
*
*       extern int uinit ( ) ;
*       extern int ureset ( ) ;
*       struct fnc_data uinit_struct =
```

17. Interrupt handlers may need to be set by the developer to insure that sm_cancel is called at all the necessary hardware and software interrupt signals. It is suggested that this setup be done either in the funclist.c routine sm_do_uinstalls, or in the function installed as an initialization function.

```
*               { 0, uinit, 0, 0, 0, 0 } ;
*       struct fnc_data ureset_struct =
*               { 0, ureset, 0, 0, 0, 0 } ;
*
* and the following lines of code, usually found near the
* beginning of the main function.
*
*       sm_install ( UINIT_FUNC, &uinit_struct, (int *)0 ) ;
*       sm_install ( URESET_FUNC, &ureset_struct, (int *)0 ) ;
*
* Note that the function installed as UINIT_FUNC is called from
* sm_initcrt() or variant, and must be installed before it is
* run. The function installed as URESET_FUNC is called from
* sm_resetcrt(), and is consequently called even if the
* application terminates abnormally, provided the correct
* signal handling is in place.
*
* Note also 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 ) ;
}
```

2.2.10

Recording and Playing Back Keystrokes

The Screen Manager provides hooks for recording and playing back keystrokes. This facility could be used to implement simple macro capabilities, or to perform regression testing on a JAM application. The developer should ensure that the record and playback functions are not in use simultaneously.

Record and playback functions are installed individually as RECORD_FUNC and PLAY_FUNC respectively.

Record/Playback Function Invocation

The record function is called from sm_getkey when it has a translated key value in hand that it is about to return to the application. The playback function is called from sm_getkey, when installed, in place of a read from the keyboard[18]. For accurate regression testing, the playback function may need to pause and flush the output to simulate a realistic rate of typing, and may need to call the asynchronous function, if there is one.

18. Since characters are recorded after processing by the key change function but played back before key change translation, some key change functions may interfere with the accurate playback of recorded keystrokes. See the description of sm_getkey in the Programmer's Reference Manual for more information.

Record/Playback Function Arguments

The record function is passed a single integer, which is the JAM logical key to record. Generally that key is recorded in some fashion for a possible playback at a later date. The playback function receives no arguments.

Record/Playback Function Return Codes

The record function should return 0. The playback function should return the logical key that was recorded at an earlier time.

Example Record/Playback System

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

```
/*
* The two functions below, record() and play(), are designed to
* 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 declarations/definitions to support installation
* of this record/playback system are shown below. They would
* typically be found in the main function source file or in
* the source file funclist.c:
*
*       extern int record ( ) ;
*       extern int play ( ) ;
*       struct fnc_data record_struct =
*               { 0, record, 0, 0, 0, 0 } ;
*       struct fnc_data play_struct =
*               { 0, play, 0, 0, 0, 0 } ;
*
* The following lines might 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 ;
*               }
*       }
*
* Note that it would be good if the main function initialized the
* variable r_time rather than counting on this record/playback
* system to do it. As it stands, the interval before the very
* first key that the user types will not be 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 ] ) ;
}
```

2.2.11

Status Line Functions

The status line function is called by the Screen Manager whenever the status line is about to be flushed, or physically written to the terminal device. It is intended for use on terminals that require unusual status line processing, beyond the scope of the generic code, but other uses are possible.

The status line function is installed individually as `STAT_FUNC`.

Status Line Function Invocation

The status line function is called when the status line is about to be physically written to the terminal display. Because of delayed write, this may or may not be at the time when the functions that specify message line text are actually called.

Status Line Function Arguments

The status line function receives no arguments. It can access copies of the text and attributes about to be flushed to the status line using the following library routine calls[19]:

```
stat_text = sm_pinquire(SP_STATLINE);
stat_attr = sm_pinquire(SP_STATATTR);
```

Status Line Function Return Codes

If the status line function returns 0, JAM continues its usual processing and actually writes out the status line. If the function returns any other value, JAM assumes that the physical write of the status line was handled in the hook function.

19. Note that `sm_pinquire`, in the case of the status text and status attribute globals, returns a pointer to a temporary copy of the arrays. These should be copied to a save location before being used.

Example Status Line Function

The following example shows the basic framework of how a status line function should probably be written:

```
/*
* This function is intended to be installed as 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.
*
* The following declarations and definitions, generally found in
* funclist.c or in the main routine source module prepare this
* routine for installation:
*
*       extern int stat ( ) ;
*       struct fnc_data stat_struct =
*               { 0, stat, 0, 0, 0, 0 }
*
* The following line of code, generally found in the funclist.c
* function sm_do_uinstalls() or in the main routine itself, is
* used to install the status line function:
*
*       sm_install ( STAT_FUNC, &stat_struct, (int *)0 ) ;
*
*/

/* Include Files */
#include "smdefs.h"     /* Screen Manager Header File */
#include "smglobs.h"    /* Screen Manager Globals */

int
stat ( )
{
        int n_columns ;         /* Width of physical display */
        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 ) ;
}
```

2.2.12

Video Processing Functions

The Screen Manager calls the developer–installed video processing function to allow for special handling of various video sequences by the application. This is a specialized hook required only when the JAM video file is unable to provide support for a particular type of terminal. Video processing functions *should not* call JAM library functions. The video processing function is called immediately before data is displayed on a JAM screen and should, therefore, perform only low-level processing.

The video processing function is installed individually as VPROC_FUNC.

Video Processing Function Invocation

The video processing function is called by JAM's output routine just before a video output operation is about to begin.

Video Processing Function Arguments

The video processing function receives two arguments. The first is an integer video processing code defined in the header file `smvideo.h` and outlined in the table below.

The second is 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 the table below. For video processing codes that require no arguments, a NULL pointer is passed.

Code	*Operation Description*	*# of params*
V_ARGR	remove area attribute	
V_ASGR	set area graphics rendition	11
V_BELL	visible alarm sequence	
V_CMSG	close message line	
V_COF	turn cursor off	
V_CON	turn cursor on	
V_CUB	cursor back (left)	1
V_CUD	cursor down	1
V_CUF	cursor forward (right)	1
V_CUP	set cursor position (absolute)	2
V_CUU	cursor up	1
V_ED	erase entire display	
V_EL	erase to end of line	
V_EW	erase window to background	5
V_INIT	initialization string	
V_INSON	set insert cursor style	
V_INSOFF	set overstrike cursor style	
V_KSET	write to soft key label	2
V_MODE4	single character graphics mode (also V_MODE5, 6)	

Code	*Operation Description*	*# of params*
V_MODE0	set graphics mode (also V_MODE1,2,3)	
V_OMSG	open message line	
V_RESET	reset string	
V_RCP	restore cursor position	
V_REPT	repeat character sequence	2
V_SCP	save cursor position	
V_SGR	set latch graphics rendition	11

Video Processing Function Return Codes

When the video processing function returns 0, JAM continues with normal processing. If it returns any other value, JAM assumes that the operation has been handled in the hook function. This allows the developer to implement only necessary operations.

Other Hook Functions

The Screen Manager provides additional hooks to handle alternative scrolling methods and block mode terminals. Those functions are best viewed as drivers, and each is described in its own chapter later in this document. Alternative scrolling is described in Chapter 10 Block mode is described in Chapter 11.

2.3
PROTOTYPED FUNCTIONS

As mentioned previously, hook function installation falls into two categories, individual installation and list installation. For those functions that are installed individually, only one of any given type can be "in use" by the application at a time, and they are called at well–defined Screen Manager events. However, the functions that are installed on function lists co–exist. There are four function lists we have discussed above, the

FIELD_FUNC list, the GROUP_FUNC list, the SCREEN_FUNC list and the CONTROL_FUNC list[20,21]. Any of those lists may have any number of functions installed simultaneously.

When a screen has a screen entry or exit function specified in the Screen Editor, that function is searched for on the SCREEN_FUNC list and generally passed two arguments: the name of the screen and the calling context bit mask. When a group has a group entry, exit, or validation function specified in the Screen Editor, that function is searched for on the GROUP_FUNC list and passed two arguments. Field entry, exit, and validation functions specified in the Screen Editor are searched for on the FIELD_FUNC list and passed four arguments, and the functions specified in control strings are searched for on the CONTROL_FUNC list and passed a single argument. The JPL `call` verb also searches the CONTROL_FUNC.

The arguments passed by default to a function on a list of a particular type are generally sufficient for the processing that is needed. However, there may be a need to customize the arguments passed to a function on a function list. Arguments passed to function list hook functions can be specified by *prototyping* the function in the function name.

2.3.1
Preparing Prototyped Functions for Installation

Functions are prototyped by appending a list of argument types to the function name in the `fnc_data` data structure used for function installation. The list of argument types is enclosed in parentheses: only character strings (abbreviated as "`s`") and integers (abbreviated as "`i`") are supported. The following example shows the definition of a `fnc_data` structure with four prototyped functions:

```
struct fnc_data func_list[] = {
        { "myfunc()", myfunc, 0, 0, 0, 0 },
        { "addinto(s,s,s)", addinto, 0, 0, 0, 0 },
        { "sm_n_putfield(s,s)", sm_n_putfield, 0, 0, 0, 0 },
        { "sm_lreset(i)", sm_lreset, 0, 0, 0, 0 }
} ;

int func_list_count = sizeof ( func_list ) /
                              sizeof ( struct fnc_data ) ;
```

20. It is critical to understand that the default field, screen, and group functions, installed as DFLT_FIELD_FUNC, DFLT_SCREEN_FUNC, and DFLT_GROUP_FUNC respectively, are not on the function lists. These default functions are installed individually.

21. There is a fifth list, SCROLL_FUNC, which is described in Chapter 10 on Alternative Scrolling Methods.

In this case, `myfunc` is prototyped to take no arguments, `addinto` to take three string arguments, `sm_n_putfield` to take two string arguments, and `sm_lreset` to take a single integer argument. Of course, when the functions are actually written, they must be defined to take the same number of arguments that is specified in the prototype. Notice, however, in the example above that the last two functions are in the **JAM** library, prototyped according to their specification in the reference manual. No code needs to be written to use them directly as hook functions. They need only to be prototyped and installed.

Functions may be listed on the PROTO_FUNC list with no prototype. In that case, **JAM** assumes that the function is written to take a single string argument. When the function is invoked, **JAM** passes a copy of the entire invoking string as an argument. In this way, screen, field, and group functions can be made to accept variable arguments in the way that control functions normally do.

JAM supports arbitrary combinations of strings and integers as function prototypes from zero to three arguments. Some combinations are supported for four, five, and six arguments, but not all. The list of all valid prototypes follows. It was chosen in such a way that most **JAM** library functions could be prototyped, if desired:

```
()

(i)          (s)

(i,i)        (s,i)        (i,s)        (s,s)

(i,i,i)      (s,i,i)      (i,s,i)      (i,i,s)
(s,s,s)      (i,s,s)      (s,i,s)      (s,s,i)

(i,i,i,i)    (s,i,i,i)    (s,s,i,i)    (s,s,s,i)
(s,s,s,s)

(i,i,i,i,i)

(i,i,i,i,i,i)
```

2.3.2

Installing Prototyped Functions

Prototyped functions may be installed on any of the function lists. However, since prototyped functions are explicitly passed arguments that match the prototype, the same prototyped function can be used by screen, group, field and control hook invocations.

To allow screen, group, field, and control hooks to share prototyped functions, there is a general purpose function list called PROTO_FUNC. It is recommended that all prototyped functions in use be installed on the PROTO_FUNC list so that they can be shared by all of the JAM function list hooks.

When the Screen Manager detects at runtime, for example, that a screen entry function was specified for the screen being opened, it first searches the SCREEN_FUNC list and then the PROTO_FUNC list for that function. Similarly for groups, the GROUP_FUNC list is searched before the PROTO_FUNC list. Field functions and control functions are searched for in the same way, first in their own function lists and then in the general–purpose PROTO_FUNC list.

2.3.3
Prototyped Function Invocation

As mentioned above, individually installed functions cannot be prototyped. Only functions appearing on function lists can be prototyped. These functions are invoked at the following times:

1. On screen, group, or field entry or exit, when a function is specified for that purpose in the Screen Editor.
2. On group or field validation, when a function is specified for that purpose in the Screen Editor.
3. When a control string starting with a caret (^) is executed.
4. When the JPL `call` verb is used.

In every case, the hook function is invoked with a line of text called a hook string, one that is often specified in the Screen Editor.

The first word of the hook string is used to search the function list for the event type and then the PROTO_FUNC list to find the function to call. The subsequent words are colon pre–processed and converted into character strings or integers based on the prototype specified if the matched function is a prototyped function[22]. Please see the relevant sections of the *Author's Guide* for more discussion of the hook processing.

It is recommended that all prototyped functions be installed on the PROTO_FUNC list.

22. If the matched function is not prototyped, and is not on the prototype function list, the arguments to be passed are fixed as described in the sections above. If the function is not prototyped, but is found on the prototyped function list, the entire invoking hook string is passed a single string argument.

2.3.4

PROTO_FUNC List Example

In the example that follows, we show how two of the functions shown as examples earlier in this document can be written and installed as prototyped functions. Both functions would be accessible for screen, field, group, and control hooks.

```
/*
* The following functions are intended as prototyped functions to
* be placed on the PROTO_FUNC function list. From that list,
* they will be accessible as control functions, screen functions,
* group functions, or from JPL code with the "call" verb.
*
* The first function, mark_flds(), takes two integer arguments.
* If the first is less than the second, all fields on the screen
* that have numeric values between the two arguments are
* temporarily highlighted. If the first argument is greater than
* the second, all fields on the screen that have numeric values
* that are not between the two fields are highlighted. The
* following string would highlight all values on the screen
* between zero and 500:
*       mark_flds 0 500
* The following string would highlight all values on the screen
* that are greater than 1000 or less than -300:
*       mark_flds 1000 -300
*
* The second function, report(), takes two string arguments. The
* first argument is the report type, and should read "field",
* "screen", "wstack" or "term". The second argument is the
* report output designation. If it is empty, the requested
* report is shown in a message window. For example, the
* following string would cause a field report to be popped 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 found in the command, the name of the
* temporary file is substituted for the tilde, otherwise the name
* is just appended at the end. The following two strings would
* each cause a screen report to be printed 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. The following string would print 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. The following string
* would cause a display terminal report to be appended to the
* file report.fil:
*       report term report.fil
*
* The following declarations and definitions support the
* installation of these functions. They are generally found
* either in the main routine source file or in the file
* funclist.c:
*
*       extern int mark_flds ( ) ;
*       extern int report ( ) ;
*       struct fnc_data proto_list[] = {
*               { "mark_flds(i,i)", mark_flds, 0, 0, 0, 0 },
*               { "report(s,s)", report, 0, 0, 0, 0 }
*       } ;
*       int proto_count = sizeof ( proto_list ) /
*                         sizeof ( struct fnc_data ) ;
*
* The following library call will install these functions. It
* is generally made either in the main function or in the
* function sm_do_uinstalls() found in the source module
* funclist.c:
*
*       sm_install ( PROTO_FUNC, proto_list, &proto_count ) ;
*/

/* Include Files */
#include "smdefs.h"     /* Screen Manager Header File */
#include "smglobs.h"    /* Screen Manager Globals */

/* Functions in this module */
int mark_flds ( ) ;
int report ( ) ;

/* 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 ) ;
}

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 -- flushed with
                call to err_reset */
            sm_mwindow ( ptr, -1, -1 ) ;
            sm_err_reset ( "Report Done..Hit Space to "
                "continue" ) ;

            /* Close message window */
            sm_close_window ( ) ;

            /* Free up the malloced buffer. */
            free ( ptr ) ;
            break ;


        default :
            /* File appended, just close it. */
            fclose ( fp ) ;
            sm_err_reset ( "File appended" ) ;
            break ;
        }
        return ( 0 ) ;
}
```

2.4
CODING STRATEGY, RULES AND PITFALLS

2.4.1
Prototyped Function Limitations

If a screen, field, group or control function is prototyped, it is not passed the standard arguments described in the previous sections. Certain of these arguments may be obtained by other methods as described below.

Accessing the Standard Arguments to Prototyped Field and Group Functions

Non–prototyped group or field functions are passed identifying information for the group or field in question. If you require access to the standard arguments that are

passed to field or group functions, but your function is prototyped, you may get some these values via the library routine `sm_inquire`. The following global variables are supported:

Mnemonic	*Meaning*
`SC_AFLDNO`	Number of the field calling a prototyped field function. Corresponds to the first standard argument to a field function.
`SC_AFLDOCC`	Occurrence number of the field calling a prototyped field function. Corresponds to the third standard argument to a field function.
`SC_AFLDMDT`	Bit mask containing contextual information about the validation state of the field and the circumstances under which a prototyped field function was called. Corresponds to fourth standard argument to a field function.
`SC_AGRPMDT`	Bit mask containing information about the validation state of the group and the circumstances under which a prototyped group function was called. Corresponds to the second standard argument passed to a non-prototyped group function.

The second standard argument to a field function, namely a pointer to a copy of the field's contents, may be obtained from `sm_getfield` or `sm_o_getfield`.

The first standard argument to a group function, a pointer to the group name, may be obtained by `sm_getcurno` and `sm_ftog` at group entry and exit. Access to the group name at group validation is not supported. since the group may be undergoing validation as part of screen validation.

Accessing the Standard Arguments to Prototyped Screen and Control Functions

Access to the standard arguments to a screen function, namely the screen name and an integer containing bit flags that specify the context of the call, is not supported if the function is prototyped. Screen functions for which this information is required cannot take advantage of function prototyping.

The standard argument to a control function, namely a pointer to a copy of the control string that invoked it, is available only if the function appears on the control or prototyped function list without a prototype. If the function has a prototype, then this information will not be available.

Passing Information to a Non–Prototyped Function

In a field validation function, if you wish to associate information with a specific field but you are not prototyping the function, and therefore cannot pass arguments to it, the memo text edits may be useful. The example below takes advantage of this feature:

```
/*
* This function is intended 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 stat_line[ 128 ] ; /* Status line 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 ( stat_line, "Invalid value %s in field. "
            "Valid values are: %s.", f_data,
            sm_edit_ptr ( f_number, MEMO1 ) + 2 ) ;

        sm_err_reset ( stat_line ) ;

        /* Return and reset cursor. */
        return ( 2 ) ;
}
```

2.4.2

Displaying Screens

There are a number of library functions provided for the display of screens as forms or windows. In general, the following rules and guidelines should be followed in choosing between them and deciding when they can be used:

- The display of screens as forms or windows from within screen functions at screen entry or screen exit is neither recommended nor supported.
- The routines `sm_jform`, `sm_jwindow`, and `sm_jclose` are provided specifically for the display and destruction of screens in applications that use the JAM Executive. Applications not using the JAM Executive should not use these routines. They are recommended over the other screen display routines in applications that do use the JAM Executive.
- The form display routine `sm_jform` manipulates the form stack appropriately. The use of any other form display routines in applications

that use the JAM Executive will exhibit unexpected behavior, as the form stack will not be synchronized with the application flow.

2.4.3
Recursion

The developer should be careful, when using hook functions, to avoid the recursion that comes from nested hook function calls. Such recursion is not easy to detect in the source code itself: some understanding of the product mechanism is required.

For example, care should be taken when writing record, playback, or key change functions that read from the keyboard, or status line functions that themselves cause the status line to be flushed. A default screen entry function that in and of itself opens new screens could be a problem.

2.4.4
Calling C Routines from JPL

The JPL `call` verb may be used to call JAM library functions, standard C functions, and developer–written hook functions. Each function to be called in this way must be installed in either the SCREEN_FUNC, CONTROL_FUNC, or PROTO_FUNC function list.

Refer to the man page for the JPL `call` verb as well as Chapter 8 of the *JPL Guide* for more information on calling C routines from JPL.

Chapter 3
Local Data Block

The Local Data Block, or LDB, is a region of memory for the storage of JAM field data that is generally shared between screens. It is discussed in the JAM *Development Overview* and in the *Author's Guide*.

3.1
LDB CREATION

The LDB is created with the library routine call `sm_ldb_init`. This routine searches for a data dictionary file created from the authoring tool with the Data Dictionary Editor. For more information about the data dictionary and the Data Dictionary Editor, see the *Author's Guide*.

If the data dictionary file is found, it is read and a single LDB entry is created in memory for every data dictionary entry that has a non–zero scope. Note that only the name of the LDB entry is placed in memory, storage for the field data that is stored with the entry is not allocated until the entry is used.

After it is created, the LDB is initialized from ASCII text files. These files, described in the *Author's Guide*, contain pairs of LDB names and values. The LDB entries named are filled with the values that follow them in the files.

3.2
HOW JAM USES THE LDB

JAM uses the LDB for the storage and propagation of field data from screen to screen in the application. Every time a screen is opened, or exposed by the closing of a window

that covers it, every field on the screen named identically to an LDB entry is filled with the value of the LDB entry. This occurs after the screen entry function is called.

Correspondingly, every time a screen is closed, or hidden when a window pops up over it, every LDB entry that is named identically to a field on the screen is filled with the value of the screen field. This occurs before the screen exit function is called.

When a screen is populated from the LDB at screen entry time, there is a subtle difference between a new screen being opened and a screen being exposed when a covering window is closed. When a screen is newly opened, only empty fields with corresponding LDB entries will be populated from the LDB. When a screen is exposed, all fields that have corresponding LDB entries will be populated.

For efficiency's sake, no LDB merge occurs when an error window is opened or closed, since these windows don't allow any data entry. Error windows are those opened with the library routine `sm_mwindow`, or one of the message functions using the `%W` pop-up window specifier.

3.3
LDB ACCESS

Data in the LDB can be accessed with the library routines `sm_n_getfield`, `sm_n_putfield`, `sm_i_getfield`, `sm_i_putfield`, and related functions that access data by field name. These routines access the data on the current screen if the field that is named exists on the current screen. If the field does not exist on the current screen, these routines access the LDB.

During screen entry and exit processing only, the search order is reversed. During the screen entry and exit functions, these access routines first search the LDB and then search the screen. This is because the LDB is merged to the screen after the screen entry function, and the screen is stored to the LDB before the screen exit function. If the search order were not reversed the data accessed would be invalid[23].

23. This could, in a very small number of cases, introduce some incompatibilities with applications that were written with earlier releases of JAM. If such compatibility problems arise, use the library function `SM_OPTION` setting the option `ENTEXT_OPTION` to `FORMFIRST`.

Chapter 4
Built–in Control Functions

This section describes control functions supplied with JAM. Note that the synopsis is for a JAM control string, not a programming language source statement. The return value of a control function can be used in a target list; see the *Author's Guide* for information on control strings and target lists.

You may use these functions in control strings and in JPL `call` statements.

jm_exit

end processing and leave the current screen

SYNOPSIS

```
^jm_exit
```

DESCRIPTION

Clears the current form or window and returns to the previous one. If the current form is the application's top–level form, JAM will prompt and exit to the operating system.

The effect is the same as the default action of the run–time system's EXIT key.

EXAMPLE

The following control string invokes a function named `process`. If it returns 0, another function is invoked to reinitialize the screen; but if it returns –1, the screen is exited. See `jm_gotop` for another example.

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

The example below shows how a form or a window can be replaced by another form or a window:

```
^(0=&w2)jm_exit
```

jm_gotop

return to application's top–level form

SYNOPSIS

```
^jm_gotop
```

DESCRIPTION

Returns to the application's top–level screen, ordinarily the first screen to appear when the application was run. All forms on the form stack and windows on the window stack are discarded.

The result is the same as the default action of the run–time system's SPF1 key.

EXAMPLE

The following menu makes use of both `jm_exit` and `jm_gotop`.

```
+-------------------------------------------------+
:                                                 :
:  Query customer database__    custquery.jam__   :
:  Update customer database_    custupdate.jam_   :
:  Free-form query__________    !sql___________   :
:  Return to previous menu__    ^jm_exit_______   :
:  Return to main menu______    ^jm_gotop______   :
:                                                 :
+-------------------------------------------------+
```

jm_goform

prompt for and display an arbitrary form

SYNOPSIS

```
^jm_goform
```

DESCRIPTION

This function pops up a window in which you may enter the name of a form; it will then close all open windows and attempt to display the form, as if that form's name had appeared in a control string. It is useful for providing a shortcut around a menu system for experienced users.

The result is the same as the default action of the run–time system's SPF3 key.

EXAMPLE

The following line, if placed in your setup file, will make the PF10 key act like SPF3 normally does:

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

jm_keys

simulate keyboard input

SYNOPSIS

```
^jm_keys keyname-or-string {keyname-or-string ...}
```

DESCRIPTION

Queues characters and function keys that appear after the function name for input to the run-time system, using `sm_ungetkey`. The run-time system then behaves as though you had typed the keys.

Function keys should be written using the logical key mnemonics listed in `smkeys.h`. Data characters should be enclosed between apostrophes `' '`, backquotes `` ` ` ``, or double quotes `" "`. This function passes its arguments to `sm_ungetkey` in reverse order, so you supply them in the natural order.

`jm_keys` will process a maximum of 20 keys. This limit includes function keys plus characters contained in strings.

EXAMPLE

Enter the name of your favorite bar, followed by a tab and the name of its owner:

```
^jm_keys 'Steinway Brauhall' TAB
^jm_keys "James O'Shaughnessy"
```

Return to the preceding menu and choose the second option:

```
^jm_keys EXIT HOME TAB XMIT
```

jm_mnutogl

switch between menu and data entry mode on a dual–purpose screen

SYNOPSIS

```
^jm_mnutogl {screen–mode}
```

DESCRIPTION

JAM supports the use of a single screen for both menu selection and data entry; one popular example is a data entry screen with a "menu bar". The screen must, however, be either one or the other at any given moment. This function switches the run–time system's treatment of the screen to the other mode. This function performs the same function as the MTGL logical key.

An optional argument may be specified which will force the screen into a particular mode, regardless of its current state. To specify menu mode, use the argument 'M' (or 'm'). To specify open–keyboard (data entry) mode, use the argument 'O' (or 'o').

jm_system

prompt for and execute an operating system command

SYNOPSIS

```
^jm_system
```

DESCRIPTION

This function pops up a small window, in which you may enter an operating system command. When you press TRANSMIT, it closes the window and executes the command. While the command is executing, your terminal is returned to the operating system's default I/O mode.

The run–time system's SPF2 key invokes this function by default.

EXAMPLE

The following line, when placed in your setup file, will cause the PF10 key to act as SPF2 normally does:

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

jm_winsize

allow end–user to interactively move and resize a window

SYNOPSIS

```
^jm_winsize
```

DESCRIPTION

Calling `jm_winsize` has the same effect as if the end–user had just hit the VWPT (viewport) logical key. The viewport status line appears and the user can move, resize and change the offset of the screen as well as move to any sibling windows. When the end–user presses XMIT the previous status line is restored.

In order for the end–user to able to move from one window to another, the windows must be siblings. Windows may be specified as siblings by specifying `&&` in a **JAM** control string. See the sections on "Viewports and Positioning" and "Control Strings" in the *Author's Guide* for further information. This function parallels the library routine `sm_winsize`.

jpl

invoke a JPL procedure

SYNOPSIS

^jpl ***procedure [argument ...]***

DESCRIPTION

This function invokes a procedure written in the JYACC Procedural Language. ***procedure*** should be the name of a JPL procedure or module; anything following that will be passed to the procedure as arguments. See the *JPL Guide* for the rules used by the JPL interpreter to determine which JPL procedure is executed. The value returned by your procedure will be returned by `jpl` for use in a target list.

This function is similar to the JPL `jpl` command. Colon expansion is done on the arguments.

EXAMPLE

The control string below invokes a JPL function to concatenate two strings and store the result in `target`.

```
^jpl concat target "king" "kong"
```

Chapter 5
Keyboard Input

Keystrokes are processed in three steps. First, the sequence of characters generated by one key is identified. Next the sequence is translated to an internal value, or logical character. Finally, the internal value is either acted upon or returned to the application ("key routing"). All three steps are table–driven. Hooks are provided at several points for application processing; they are described in the chapter "Writing and Installing Hook Functions".

5.1
LOGICAL KEYS

JAM processes characters internally as logical values, which frequently (but not always) correspond to the physical ASCII codes used by terminal keyboards and displays. Specific physical keys or sequences of physical keys are mapped to logical values by the key translation table, and logical characters are mapped to video output by the MODE and GRAPH commands in the video file. For most keys, such as the normal displayable characters, no explicit mapping is necessary. Certain ranges of logical characters are interpreted specially by JAM; they are

- `0x0100 to 0x01ff`: operations such as tab, scrolling, cursor motion
- `0x6101 to 0x7801`: function keys PF1 – PF24
- `0x4101 to 0x5801`: shifted function keys SPF1 – SPF24
- `0x6102 to 0x7802`: application keys APP1 – APP24

5.2 KEY TRANSLATION

The first two steps together are controlled by the key translation table, which is loaded during initialization. The name of the table is found in the environment (see the *Configuration Guide* for details). The table itself is derived from an ASCII file which can be modified by any editor; a screen-oriented utility, `modkey`, is also supplied for creating and modifying key translation tables (see the *Utilities Guide*).

The algorithm described below assumes the default case, where you have not specified a timing interval with the `KBD_DELAY` entry in the video file.

JAM assumes that the first character of any multi-character key sequence to be translated to a single logical key is a control character in the ASCII chart (0x00 to 0x1f, 0x7f, 0x80 to 0x9f, or 0xff). All characters not in this range are assumed to be displayable characters and are not translated.

Upon receipt of a control character, the keyboard input function `sm_getkey` searches the translation table. If no match is found on the first character, the key is accepted without translation. If a full match is found on the first character, an exact match has been found, and `sm_getkey` returns the value indicated in the table. The search continues through subsequent characters until either

1. an exact match on n characters is found and the n+1'th character in the table is 0, or n is 6. In this case the value in the table is returned.
2. 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.

This last step is of some importance: if the operator presses a function key that is not in the table, the Screen Manager must know "where the key ends". The algorithm used is as follows. The table 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 no entry matches by type at the n'th character, the shortest sequence that matches on n–1 characters is used.) This method allows `sm_getkey` to distinguish, for example, between the sequences `ESC O x`, `ESC [ A`, and `ESC [ 1 0 ~`.

If you do have a `KBD_DELAY` entry in the video file, you may specify key sequences in the key translation file that are substrings of other sequences. For example, the sequences `Esc` and `Esc [ C` could both have logical values, even though one is a substring of the other. In this case, **JAM** waits up to the specified timing interval between processing characters to determine if a character is a single keystroke or belongs to a combination sequence. Refer to the *Configuration Guide* for details.

5.3 KEY ROUTING

The main routine for keyboard processing is `sm_input`. This routine calls `sm_getkey` to obtain the translated value of the key. It then decides what to do based on the following rules.

If the value is greater than 0x1ff, `sm_input` returns to the caller with this value as the return code.

If the value is between 0x01 and 0x1ff, the key is first translated via the key translation table. This table is changed with the library routine `sm_keyoption`. Then processing is determined by a routing table. Use `sm_keyoption` to get and set the routing information for a particular key. The routing value consists of two bits, examined independently, so four different actions are possible:

1. If neither bit is set, the key is ignored.
2. If the EXECUTE bit is set and the 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.
3. If the RETURN bit is set, `sm_input` returns the logical value to the caller; otherwise, `sm_getkey` is called for another value.
4. If both bits are set, the key is executed and then returned.

The default settings are *ignore* 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 application function keys (PF1–24, SPF1–24, APP1–24, 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.

Applications can change key actions on the fly by using `sm_keyoption`. For example, to disable the backtab key the application program would execute

```
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)
```

Key mnemonics can be found in the file `smkeys.h`.

Chapter 6
Terminal Output Processing

JAM uses a sophisticated *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 being solicited or the library function `sm_flush` has been called. Instead, the run–time system does screen updates in memory, and keeps track of the display positions thus "dirtied". Flushing begins when the keyboard is opened; but if you type a character while flushing is incomplete, the run–time system will process it before sending any more output to the display. This makes it possible to type ahead on slow lines. You may force the display to be updated by calling `sm_flush`.

JAM takes pains to avoid code specific to particular displays or terminals. To achieve this it defines a set of logical screen operations (such as "position the cursor"), and stores the character sequences for performing these operations on each type of display in a file specific to the display. Logical display operations and the coding of sequences are detailed in the video file section of the JAM *Configuration Guide*; the following sections describe additional ways in which applications may use the information encoded in the video file.

6.1
GRAPHICS CHARACTERS AND ALTERNATE CHARACTER SETS

Many terminals support the display of graphics or special characters through alternate character sets. Control sequences switch the terminal among the various sets, and characters in the standard ASCII range are displayed differently in different sets. JAM supports alternate character sets via the `MODEx` and `GRAPH` commands 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 will first transmit the sequence that switches to mode, then transmit `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.

6.2
THE STATUS LINE

JAM reserves one line on the display for error and other status messages. Many terminals have a special status line (not addressable with normal cursor positioning); if such is not the case, JAM will use the bottom line of the display for messages. There are several sorts of messages that use the status line; they appear below in priority order.

1. Transient messages issued by `sm_err_reset` or a related function
2. Ready/wait status
3. Messages installed with `sm_d_msg_line` or `sm_msg`
4. Field status text
5. Background status text

There are several routines that display a message on the status line, wait for acknowledgement from the operator, and then reset the status line to its previous state: `sm_query_msg`, `sm_err_reset`, `sm_emsg`, `sm_quiet_err`, and `sm_qui_msg`. `sm_query_msg` waits for a yes/no response, which it returns to the calling program; the others wait for you to acknowledge the message. These messages have highest precedence.

`sm_setstatus` provides an alternating pair of background messages, which have next highest precedence. Whenever the keyboard is open for input the status line displays Ready; it displays `Wait` when your program is processing and the keyboard is not open. The strings may be altered by changing the `SM_READY` and `SM_WAIT` entries in the message file.

If you call `sm_d_msg_line`, the display attribute and message text you pass remain on the status line until erased by another call or overridden by a message of higher precedence.

When the status line has no higher priority text, the Screen Manager checks the current field for text to be displayed on the status line. If the cursor is not in a field, or if it is in a field with no status text, JAM looks for background status text, the lowest priority. Background status text can be set by calling `sm_setbkstat`, passing it the message text and display attribute.

In addition to messages, the rightmost part of the status line can display the cursor's current screen position, as, for example, `C 2,18`. This display is controlled by calls to `sm_c_vis`.

During debugging, calls to `sm_err_reset` or `sm_quiet_err` can be used to provide status information to the programmer without disturbing the main screen display. Keep in mind that these calls will work properly only after screen handling has been initialized by a call to `sm_initcrt`. `sm_err_reset` and `sm_quiet_err` can be called with a message text that is defined locally, as in:

```
sm_err_reset("ZIP CODE INVALID FOR THIS STATE.");
```

However, the JAM library functions use a set of messages defined in an internal message table. This table is accessed by the function `sm_msg_get`, using a set of defines in the header file `smerror.h`. For example:

```
sm_quiet_err (sm_msg_get (SM_MALLOC));
```

The message table is initialized from the message file identified by the environment variable SMMSGS. Application messages can also be placed in the message file. See the section on message files in the *Configuration Guide*.

Chapter 7
Writing International (8 bit) Applications

7.1
INTRODUCTION

This chapter describes how to use the 8 bit internationalization capabilities that have been incorporated into JAM Release 5.

From the point of view of someone who has used JAM without these features, a few differences will be apparent immediately. Other, more subtle, differences will emerge as the package is used in building language–independent applications. Finally, many of the changes were made so that the development utilities could be localized for use in other countries. These will largely go unnoticed by people using the package in English.

7.1.1
General Overview

JAM's support for 8 bit international character sets allows JAM and applications created with with it to be "localized" for use in non–English–speaking countries. This means that an application can be made to look like it originated in 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. Notwithstanding this, many of the features that are only visible to programmers continue to be in English since many programmers are used to working in English.

The capabilities described here are limited to languages in which characters can be represented in 8 bits of information and those that use a left–to–right entry order. This eliminates the complexities associated with many far– and middle–eastern languages.

7.2 LOCALIZATION

JAM and JAM applications can be localized by taking the following steps:

- Use the Screen Editor to translate all screens in the application.
- Translate and recompile the message file.
- Translate the documentation.

7.2.1 Background

JAM was originally developed with some internationalization issues in mind. It has always used 8 bit character data, without appropriating a bit for internal use. So one of the major demands of the international market was already satisfied.

Date and time formats have always been completely specified by the screen creator. The wide variety of formats available in Release 4 could satisfy most requirements. In Release 5, additional capabilities were added to make it easier to convert screens from one language to another. Currency formats were the least international of the features in the Release 4 product. Release 5 makes these completely language independent.

Each of the sections below discusses some aspect of internationalization.

7.2.2 8 Bit Character Data

As pointed out in the introduction, JAM supports 8 bit character data. Video files specific to the terminal can give special instructions, if necessary, as to 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 (e.g. 0x01 – 0x1f, or 0x80 – 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. The cursor control keys (arrows, tab, etc.), function keys, and soft 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 will 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, `modkey` (or a text editor) can be used 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.

If you have questions about how to display non–English characters or to receive them from the keyboard, consult Chapter 5 on keyboard input and Chapter 6 on terminal output (video processing).

7.2.3
Date and Time Fields

Date and time fields have been completely revamped in Release 5. They have been combined to enable one field to have both date and time information. This, and the fact that more flexibility was added to date and time formatting, required changes to the date and time mnemonics. For example, in Release 4, the mnemonic `mm` was used for a 2–digit month in date fields as well as the specifier for minutes in time fields. Clearly, this cannot serve both purposes when the fields are combined.

In Release 5, the mnemonics for specifying date and time formats are stored in the message file so they may be changed. In addition, they are stored in a "tokenized" form internally which provides two major benefits. First, the need to parse the formats at runtime is eliminated, thus speeding up processing and reducing memory requirements. Second, screen designers in different countries editing the same screen will all see date and time specifications in formats they are used to. For example, if an English screen designer created a date field with the format `mon/day/year`, it might show up on a French system as `mois/jour/annee`.

The problem of interchanging the month and day is dealt with later.

The table below shows the default message file entries for date and time mnemonics:

Msg # Mnemonic	*Date/Time Mnemonic*	*Tokenized Format*	*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	abbrev. month name
FM_MONL	MONL	%*m	long month name

Thus, a date field specified as `mm/dd/yyyy` in Release 4 would be `MON2/DATE2/YR4` in Release 5. The `f4to5` conversion program will convert the format to `%m/%d/%4y` internally so it will automatically show up correctly when the screen is edited. The mnemonics were chosen to correspond to ANSI standards. You can change them to suit your own needs by simply changing the message file and running `msg2bin`. To change the mnemonic for a 4 digit year from `YR4` to `YYYY`, for example, change the message file line

```
FM_YR4 = YR4
```

to

```
FM_YR4 = YYYY
```

and run `msg2bin`.

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. What is important, however, is the ability to modify an application to operate in a different language. The goal is that only the text of the screens and the message file should need to be changed.

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

```
Mon Apr 4, 1989
```

whereas on a French system it would be

```
Lun Avr 4, 1989
```

This happens without changing the date format. All that has changed are the names and abbreviations of the months and days which are also stored in the message file so it is a simple matter to convert them.

Now consider a date field which in English should show up in `mm/dd/yyyy` form but should appear in French as `dd-mm-yyyy`. In this case, the date format itself would have to be modified. For this reason, 10 additional formats are supplied for the designer's use. For instance, in the message file the designer can specify a new date mnemonic called `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 screen, needs to be changed to convert the date field to French.

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.

The following table shows the default message file entries for these extra date mnemonics:

Msg Number Mnemonic	*Date/Time Mnemonic*	*Tokenized Form*	*Corresponding Msgfile Entry*	*Default*
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
FM_3MN_DEF_DT	DEFAULT3	%3f	SM_3DEF_DTIME	%m/%d/%2y %h:%0M
FM_4MN_DEF_DT	DEFAULT4	%4f	SM_4DEF_DTIME	%m/%d/%2y %h:%0M
FM_5MN_DEF_DT	DEFAULT5	%5f	SM_5DEF_DTIME	%m/%d/%2y %h:%0M
FM_6MN_DEF_DT	DEFAULT6	%6f	SM_6DEF_DTIME	%m/%d/%2y %h:%0M
FM_7MN_DEF_DT	DEFAULT7	%7f	SM_7DEF_DTIME	%m/%d/%2y %h:%0M
FM_8MN_DEF_DT	DEFAULT8	%8f	SM_8DEF_DTIME	%m/%d/%2y %h:%0M
FM_9MN_DEF_DT	DEFAULT9	%9f	SM_9DEF_DTIME	%m/%d/%2y %h:%0M

Thus, if the screen designer specifies a date field with the format DEFAULT DATE, it would show up in mm/dd/yy form. If the line

```
SM_1DEF_DTIME = %m/%d/%2y
```

in the message file were changed to

```
SM_1DEF_DTIME = %d-%m-%2y
```

the date would show up in dd-mm-yy form. To change the mnemonic for this date format to REGULAR DATE, the message FM_1MN_DEF_DT should be modified.

7.2.4

Currency Fields

Like date and time fields, currency fields have been modified in Release 5. Since it is not uncommon in Europe to be dealing with several currencies simultaneously, release 5 does not force any one system on the screen creator. Thus, the formatting capabilities were enhanced to support any convention the screen creator might desire. As with date and time formats, a "default" format is supplied that causes the actual format to be taken from the message file. For currency fields however, this option is supplied only for the parts of the format that may vary from one currency to another.

The new release allows the following items to be specified for currency fields:

- the decimal symbol (usually dot or comma)
- minimum number of decimal places
- maximum number of decimal places
- thousands separator (usually dot or comma; b = blank)
- currency symbol to be used (up to 5 characters)
- placement of currency symbol (left, right or at decimal pt)
- default currency from the message file (to replace the above entries)
- rounding (round–up, round–down, round–adjust)
- fill character
- justification
- clear if zero
- apply if empty

There is a slight problem in specifying currency symbols when using the Screen Editor. Since the currency symbol is entered into a regular field, it is not possible to enter trailing spaces (they are always stripped off). Thus, to specify a leading currency symbol separated from the data by a space (`FF 123.456,78`) you must use the message file. For this reason, the dot (.) may be used to signify a space when entered into the currency field. A dot in the message file for this purpose will appear as a dot.

The default currency formats are strings of the form ***rmxtpccccc*** where:

- ***r*** = decimal symbol (usually comma or dot)
- ***m*** = minimum number of decimal places
- ***x*** = maximum number of decimal places

- *t* = thousands separator (usually comma or dot; b = blank)
- *p* = placement of currency symbol (l, r or m)
- *ccccc* = up to 5 characters for the currency symbol

Thus, if the screen designer specifies a currency field with the format CURRENCY, it would show up in `$999,999.99` form. If the line

```
SM_0DEF_CURR = ".22,1$"
```

in the message file were changed to

```
SM_0DEF_CURR = ",22.1FF"
```

the field would show up as `FF 999.99,99`. To change the mnemonic for this currency field, the message FM_0MN_CURRDEF should be modified. The following table shows the default message file entries for the currency mnemonics:

Msg Number Mnemonic	*Currency Mnemonic*	*Corresponding Msgfile Entry*	*Default*
FM_0MN_CURRDEF	CURRENCY	SM_0DEF_CURR	.22,1$
FM_1MN_CURRDEF	NUMERIC	SM_1DEF_CURR	.09,
FM_2MN_CURRDEF	PLAIN	SM_2DEF_CURR	.09
FM_3MN_CURRDEF	DEFAULT3	SM_3DEF_CURR	.09
FM_4MN_CURRDEF	DEFAULT4	SM_4DEF_CURR	.09
FM_5MN_CURRDEF	DEFAULT5	SM_5DEF_CURR	.09
FM_6MN_CURRDEF	DEFAULT6	SM_6DEF_CURR	.09
FM_7MN_CURRDEF	DEFAULT7	SM_7DEF_CURR	.09
FM_8MN_CURRDEF	DEFAULT8	SM_8DEF_CURR	.09
FM_9MN_CURRDEF	DEFAULT9	SM_9DEF_CURR	.09

7.2.5
Decimal Symbols

JAM accommodates 3 decimal symbols which are used in different circumstances:

- System Decimal Symbol
- Local Decimal Symbol
- Field Decimal Symbol

The System Decimal Symbol is the one that C library routines like `atof` and `sprintf` use. The Local Decimal Symbol is the one that is used when local customs are followed (dot in English; comma in French). The Field Decimal Symbol is the one specified for a given field if that field is not observing local conventions.

The System and Local Decimal Symbols are obtained from the operating system if the operating system supports such things (see the JAM *Installation Guide* for your operating system). The Local Decimal Symbol may be specified in the message file (message `SM_DECIMAL`), in which case it overrides the operating system decimal symbol. Dot is the system decimal if no symbol is specified in the message file and if the operating system does not supply one.

The sections below describe the circumstances under which each of the different symbols is used.

7.2.6
Character Filters

The one time that JAM requires some knowledge of the meaning of the data is while enforcing the character filters on a field. The filters currently supported are digits only, numeric, alphabetic, alphanumeric, yes/no, 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 in-

formation 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, the expression

```
[a-z]*
```

will match only the English lower case letters. The European character ä, for example, is not matched by this expression.

7.2.7
Status and Error Messages

All messages produced by JAM are stored in the message file so they may be easily localized. Each message is a complete phrase or sentence. Message components are never pieced together because doing so would make it difficult to translate to a language that has a sentence structure different from English.

7.2.8
Screens in the Utilities

These screens were memory resident in Release 4. For international customers they must be modifiable.

A linkable `jxform` is provided, and the library containing the source for the screens is made available. A developer may translate the screens and relink the utilities. Similarly `modkey` is developer–linkable, and the source for its screens is provided. In this way the screens remain memory resident and no compromise of speed need be made.

Unfortunately this solution is not ideal if several users on the same machine wish to use different languages. To support this, the screens may be kept on disk. The current mechanism of `SMPATH` allows run–time selection of the set of screens to be used.

7.2.9

Screens in Application Programs

The same approach discussed in the section on screens in the utilities can be used for screens in application programs. Thus different language screens can be kept in separate directories and the user can specify which is to be used at run–time.

7.2.10

Menu Processing

`sm_input` returns the first character of the selected entry. This, of course, is not language independent. **JAM** utilities have been modified to use the current field number rather than the return value. Because it cannot be assumed that all entries will have unique first letters, the `string` option is specified.

Application programs intended for an international market should not rely on the initial character of the menu selection. The field number containing the cursor is a better way of determining which selection the operator has made. However the field numbers may change if the screen is redesigned. Note that this is not a problem when the **JAM** Executive is used, since the **JAM** Executive uses relative field numbers to determine the control string to execute when a menu field is selected.

A new edit was instituted in **JAM** 4 that specifies the return code from a `return entry` (or `menu`) field. The screen creator specifies the return code (an integer) when designing the screen. If this edit exists, `sm_input` uses that value as the return code to the calling program. If this edit does not exist, the usual return code is used.

7.2.11

`lstform`, `lstdd`, and `jammap`

These utilities list data about the screen in English. Since they are often used for documentation it is important that the text be translatable to other languages. Thus the textual material, headings, etc., have been moved to the message file.

7.2.12

Range Checks

Range checks for numeric data are presently correctly handled since they use the C library routine `atof` (assuming that the "strip" routine works properly).

Alphabetic data presents special problems. One of the major issues for internationalization is the collating sequence of a language. For dictionary or telephone book proces-

sing the problem is particularly troublesome. For example, upper and lower case letters compare equal. Also, in a telephone book, `St.` and `Saint` compare equal, 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.

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.

For these reasons no changes have been made to the Release 4 method of range checking. `strcmp` and `memcmp` continue to be used. These compare the internal values of the characters, without regard to their meanings in the local language.

7.2.13

Calculations Using `@SUM` and `@DATE`

These keywords have been retained even though they are language specific. Computations with dates assume the Gregorian calendar. No provision is made for other calendars.

7.2.14

`sm_dblval` and `sm_dtofield`

These routines use the C library routines `atof` and `sprintf` therefore correctly interpret the System Decimal Symbol (radix character).

7.2.15

`sm_is_yes` and `sm_query_msg`

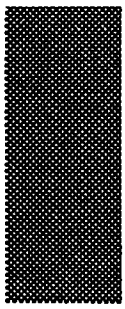
These routines use the characters in the message file for `y` and `n` and thus are already internationalized. They use `toupper` to recognize the upper case variations.

7.2.16

Batch Utilities

All the utility messages, including usage messages are in the message file.

The mnemonics for logical keys (XMIT, EXIT, etc.) 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.



Chapter 8
Writing Portable Applications

The following section describes features of hardware and operating system software that can cause JAM to behave in a non–uniform fashion. An application designer wishing to create programs that run across a variety of systems needs to be aware of these factors.

8.1
TERMINAL DEPENDENCIES

JAM can run on display terminals of any size. On terminals without a separately addressable status line, JAM will steal the bottom line of the display (often the 24th) for a status line, and status messages will overlay whatever is on that line. A good lowest common denominator for screen sizes is 23 lines by 80 columns, including the border (21 if two–line soft key labels will be used).

Different terminals support different sets of attributes. JAM makes sensible compromises based on the attributes available; but programs that rely extensively on attribute manipulation to highlight data may be confusing to users of terminals with an insufficient number of attributes. Colors will not show up on monochrome terminals, e.g.—use of graphics character sets is particularly terminal dependent.

Attribute handling can also affect the spacing of fields and text. In particular, anyone designing screens to run on terminals with onscreen attributes must remember to 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).

The key translation table mechanism supports the assignment of any key or key sequence to a particular logical character. However, the number and labelling of function

keys on particular keyboards can constrain the application designer who makes heavy use of function keys for program control. The standard VT100, for instance, has only four function keys. For simple choices among alternatives, menus are probably better than switching on function keys.

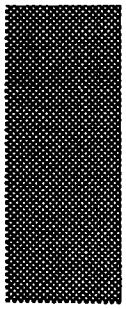
Using function key labels, or keytops, instead of hard–coded key names is also important to making an application run smoothly on a variety of terminals. Field status text and other status line messages can have keytops inserted automatically, using the `%K` escape. No such translation is done for strings written to fields; in such cases, you may want to place the strings in a message file, since the setup file can specify terminal–dependent message files.

8.2
ITEMS IN `smmach.h`

The header file `smmach.h`, which is supplied with the JAM library, contains information that library routines need to deal with certain machine, operating system, and compiler dependencies. These include:

- The presence of certain C header files and library functions.
- Byte ordering in integers and support for the unsigned character type.
- Path name and command line argument separator characters.
- Pointer alignment and structure padding.

The header file is thoroughly commented, and application designers are encouraged to make use of the information there.



Chapter 9
Writing Efficient Applications

9.1
MEMORY–RESIDENT SCREENS

Memory–resident screens are much quicker to display than disk–resident screens, since 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 (see the *Utilities Guide*), then compiled and linked with the application program.

`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 will generally have the same name as the file the original screen was originally stored in.

A more flexible way of achieving the same object is to use a memory–resident screen list. Bear in mind that the JAM Screen Editor can only operate on disk files, so that altering memory–resident screens during program development requires a tedious cycle of test – edit – reinsert with `bin2c` – recompile. The JAM library maintains an internal list of memory–resident screens that `sm_r_window` and related functions examine. Any screen found in the list will be displayed from memory, while screens not in the list will be sought on disk. This means that the application can be coded to use one set of calls, the r–version, and screens can be configured as disk– or memory–resident simply by altering the 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 would use code like the following:

```
#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);
```

Note the last entry in the screen list: 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 your list to JAM's internal list.

Using memory–resident screens (and configuration files, see the next section) is, of course, a space–time tradeoff: increased memory usage for better speed.

JAM will append the extension found in the setup variable `SMFEXTENSION` to screen names (*e.g.* in control fields) that do not already contain an extension; you must take this into account when creating the screen list. JAM may also convert the name to uppercase before searching the screen list; this is governed by the `SMFCASE` variable.

9.2
MEMORY–RESIDENT CONFIGURATION FILES

Any or all of the three configuration files required by JAM can be made memory resident. First a C source file must be created from the binary version of the file, using the `bin2c` utility; see the *Utilities Guide*. The source files created are not readily decipherable. The following fragment 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_initcrt ("");

/* ...possibly initialize function and
 * form lists  */

/* ...application code */
```

If a file is made memory–resident, the corresponding environment variable or SMVARS entry can be dispensed with.

9.3
MEMORY–RESIDENT KEYSETS

Keysets may be made memory–resident by converting them to source language structures via `bin2c` or related utilities, installing them via `sm_formlist`, and compiling and linking them into your executable.

9.4
MESSAGE FILE OPTIONS

If you need to conserve memory and have a large number of messages in message files, you can make use of the MSG_DSK option to `sm_msgread`. This option avoids loading the message files into memory; instead, they are left open, and the messages are fetched from disk when needed. Bear in mind that this uses up additional file descriptors, and that buffering the open file consumes a certain amount of system memory; you will gain little unless your message files are quite large.

9.5
MEMORY–RESIDENT JPL

JPL "load", "public" and "memory–resident" modules may 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.

Load and public modules may alternatively exist as files or in libraries. Refer to section 3.1 in the *JPL Guide* 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.

9.6
JPL VS. COMPILED LANGUAGES

JPL code execution goes through an extra layer of interpretation that compiled code, such as C, does not. In most cases, the total run time is too small to matter, but if a JPL function is long or loops many times and a delay is noted, it may pay to rewrite it in C.

See also Chapter 9 of the *JPL Guide* for hints on improving the performance of your JPL.

9.7
AVOIDING UNNECESSARY SCREEN OUTPUT

Several of the 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. To take an example: JAM can do all its cursor positioning using the CUP (absolute cursor position) command. However, it will use the 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 is capable of saving and restoring the cursor position itself (SCP, RCP), JAM will use those sequences instead of the more verbose CUP.

The global variable `I_NODISP` may also be used to decrease screen output. While this variable is set to 0 (via `sm_iset`), calls into the JAM library will cause the internal screen image to be updated, but nothing will be written to the actual display; the display can be brought up to date by resetting `I_NODISP` to 1 and calling `sm_rescreen`. With the implementation of delayed write this sort of trick is rarely necessary.

9.8
STUB FUNCTIONS

Certain Screen Manager 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 the Screen Manager libraries not be pre–linked or pre–bound, *i.e.* is not supported on all systems. The following facilities may be stubbed out:

Subsystem	*#define*
the math package	NOCALC
scrolling functions	NOSCROLL
time and date functions	NOTIMEDATE
help screens	NOHELP
shifting fields	NOSHIFT
range checking functions	NORANGE
word wrap	NOWRAP
field zoom expansion	NOZOOM
regular expressions	NOREGEXP
form libraries	NOFORMLIB
JYACC procedural language	NOJPL
Runtime JPL compiler (see note)	NOJPLCOMP
read/write data structure	NOSTRUCT
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

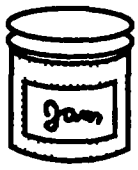
To omit any one or combination of the above, first `#define` the appropriate macro in your application, then `#include` the stubs file. This must only be done once, preferably in the application's main routine source file. For example, if the application is not going to use scrolling fields, the scrolling functions could be omitted, and the application source might look like the following:

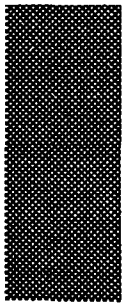
```
#define NOSCROLL
#include "smdefs.h"
#include "smstubs.c"

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 the Screen Manager library to the application. The bulk of the savings will be in code space. The stubbing technique does not work on systems where the library is itself a linked entity, such as a shareable 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, load, and public modules may still be used, but they must be compiled with `jpl2bin`.





Chapter 10
Alternative Scrolling

By default, the 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: in memory, on disk, or wherever desired. In this case, the application must install a *scrolling driver* which will be called by JAM with an interface defined by JAM; this replaces JAM's default scroll driver. The driver will be called to initialize the array, get and put occurrences, etc.

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.

10.1
USING ALTERNATIVE SCROLLING

The general procedure for using alternative scrolling in an application is as follows:

1. Write a scroll driver according to the specifications to be described later.
2. Install the scroll driver. See the chapter "Writing and Installing Hook Functions" for general information on installation. Note that scroll drivers are installed in a function list as `SCROLL_FUNC`. A default driver may be installed individually as `DFLT_SCROLL_FUNC`. In addition, `sm_alsc_init` must be called prior to use of a scroll driver. The easiest way to do this is to define the macro `ALT_SCROLLING` to be `1` inside of `jmain.c` and/or `jxmain.c`.

3. For each field which is to use the scroll driver, enter the name of the driver (as specified in the installation procedure in the previous step) as the field's `Alternative scrolling method`, via the Screen Editor. The `Alternative scrolling method` is entered in the field size screen of the Screen Editor.

A sample driver that uses disk–based scrolling, `sm_fb_sc`, is provided with JAM. It is distributed in source form in the file `fbscr.c`, and in object form in the Screen Manager library. It must be installed to be used.

Another sample driver, `sm_ldb_sc`, is provided only in object form in the JAM Executive library. It uses the local data block to store occurrences. It is useful only when a scrolling array is both on a screen and in the LDB. Normally, JAM allocates buffers to hold both the screen occurrences and the LDB occurrences. `sm_ldb_sc` enables JAM to avoid allocating additional buffers to hold the screen occurrences. `sm_ldb_sc` is not distributed in source form, since it makes use of internal JAM functions to perform its task.

10.2
WRITING A SCROLL DRIVER

When a screen with a scrolling array is displayed, the scroll driver will be called so that JAM can fill the onscreen portion of the array with data, if any. If the user or program scrolls new data onscreen, the driver will be called to retrieve it. If data is changed and then scrolled offscreen or offscreen data is changed directly, the driver is sent the new data so it can keep the array up to date. Finally, when the screen is made inactive, JAM updates the LDB from the driver, as necessary.

The driver routine must be written according to the following specifications.

JAM calls the driver, never vice versa. It is passed a pointer to a `struct alt_scroll` (defined in `smaltsc.h`, which should be included) whose elements are:

```
struct alt_scroll
{
    VOIDPTR as_scid;    /* Application supplied field handle */
    char *as_buffer;    /* Buffer Containing Information */
    int as_type;        /* Type of function requested */
    int as_occur;       /* Meaning varies according to as_type */
    int as_number;       /* Meaning varies according to as_type */
    int as_size;        /* Maximum size of an occurrence */
    int as_retval;       /* Return value */
};
```

The possible values of `as_type` as follows:

AS_INIT_FUNC	Initialize a scrolling field
AS_CLEAR_FUNC	Delete all occurrences
AS_GDATA_FUNC	Get field data of an occurrence (from the driver)
AS_PDATA_FUNC	Put (update) an occurrence (to the driver)
AS_INSRT_FUNC	Insert blank lines
AS_DLT_FUNC	Delete a range of lines
AS_NMUSD_FUNC	Return number of largest non–blank occurrence
AS_GTSPC_FUNC	Change number of allocated occurrences
AS_RLS_FUNC	Release (de–initialize) a scrolling field

Of these function types, only AS_INIT_FUNC and AS_GDATA_FUNC need be supported by the driver. If AS_PDATA_FUNC is not supported, the field should be protected from modification (clearing and data entry). The driver should return 0 if a function is supported, –1 if not.

For any given field, the driver is first called with AS_INIT_FUNC and last with AS_RLS_FUNC. Any number of calls with the other function types may intervene. In AS_INIT_FUNC only, the field is identified by name and number. Thereafter it is identified by a pointer to an area set up by the driver.

JAM sets the structure member `as_scid` to 0 when it calls `AS_INIT_FUNC`. `AS_INIT_FUNC` should set `as_scid` to a field identifier value, usually a pointer to a field structure. The other functions (`AS_CLEAR_FUNC`, etc.) must use `as_scid` to access the field since they do not have access to the field name or number.

The structure member `as_size` always contains the maximum shifting length of the field.

The meanings of the other members of `struct alt_scroll` vary according to the function type and are described below.

When `as_type` is AS_INIT_FUNC:

- `as_number` contains the field number of the scrolling field.
- `as_buffer` contains a pointer to the name of the field or NULL if it does not have one
- `as_occur` contains the maximum number of occurrences for that scrolling field.
- `as_retval` should be set to 0 indicating success or –1 indicating that alternative scrolling methods are not available for this field for some reason. If the access function returns a value less than zero or `as_retval` is set to a non–zero value the Screen Manager will attempt to use the standard scrolling method.

When `as_type` is AS_CLEAR_FUNC:

- as_number, as_buffer and as_occur are not currently meaningful.
- as_retval should be set to 0.

When as_type is AS_GDATA_FUNC:

- as_occur is the occurrence number to be retrieved
- as_buffer contains a pointer to where the data should be placed; data must be in the format described below AS_PDATA_FUNC.
- as_number is not currently meaningful.
- as_retval should be set to 0 indicating a successful retrieval, or non–zero indicating failure.

When as_type is AS_PDATA_FUNC:

- as_occur is the number of the occurrence being sent to the driver.
- as_buffer contains a pointer to the data being sent; this includes field data as well as other information. The pointer will no longer be valid after the driver returns.
- as_number is not currently meaningful.
- as_retval should be set to 0 indicating success or non–zero failure.

NOTE: The format of as_buffer for AS_GDATA_FUNC and AS_PDATA_FUNC: as_buffer contains binary data of length given by as_size. (It may contain null bytes and it will not be null terminated.) Generally the driver will store the data without examining or changing it. If it must be examined the access function sm_atrans must first be called. This will convert it to and from an application readable structure. It works as follows:

```
int sm_astrans(struct alt_scroll *, struct altsc_trans *, int
get_or_put);

struct altsc_trans
{
    unsigned short at_attr;    /* Scrolling attributes (0 if none) */
    char at_val1;      /* Currently only MDT and VALIDED are used */
    char at_buffer[256];   /* Field data string */
};
```

If "get_or_put" is AT_GET, converts from alt_scroll.as_buffer to alt_trans structure; if AT_PUT, converts in the reverse direction.

When as_type is AS_INSRT_FUNC:

- as_number contains the number of blank occurrences to be inserted.

- as_occur is the number of the occurrence before which the occurrences should be inserted.
- as_buffer is not currently meaningful.
- as_retval should be set to 0 indicating that the records were successfully inserted. If no records were successfully inserted, as_retval should be set to a negative number. If less than the requested number were inserted, as_retval should be set to that number.

When as_type is AS_DLT_FUNC:

- as_occur is the number of the first occurrence to be deleted.
- as_number contains the number of occurrences to be deleted.
- as_buffer is not currently meaningful.
- as_retval should be set to 0 to indicate that all the records requested were successfully deleted. If no records could be deleted, as_retval should be set to a negative number. If less than the requested number were deleted, as_retval should be set to that number.

When as_type is AS_NMUSD_FUNC:

- as_number contains the largest occurrence for which the Screen Manager has instructed the access method to allocate space, and thus the largest which could possibly be non–blank.
- as_buffer and as_occur are not currently meaningful.
- as_retval should be set to the number of the largest non blank occurrence, 0 if all occurrences are blank, and less than zero if the number could not be determined.

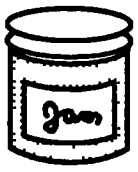
When as_type is AS_GTSPC_FUNC:

- as_occur contains the total number of occurrences for which space should be allocated; most systems will simplify this by allocating the maximum necessary space in AS_INIT_FUNC (the value of the structure member as_occur in AS_INIT_FUNC).
- as_number is not meaningful.
- as_buffer contains the data with which to initialize the new occurrences.
- as_retval should contain the number of occurrences for which space was actually allocated.

NOTE: The number of occurrences in this function is the total, not the additional.

When `as_type` is AS_RLS_FUNC:

- `as_retval`, `as_number`, `as_buffer`, and `as_occur` are not currently meaningful.



Chapter 11
Block Mode

This chapter describes how to use JAM's block mode capabilities from an end user's point of view, and how to develop a block mode driver for JAM from a developer's point of view.

11.1
USING BLOCK MODE

11.1.1
General Overview

The purpose of the block mode interface is to allow JAM to be used with terminals, like the HP2392A and IBM 3270's, that operate in block mode. Such terminals, which are hereinafter referred to as block mode terminals, operate differently than their interactive or character mode counterparts in that they do not interact with the computer on every keystroke. Instead, a formatted screen is sent to the terminal and processed by the terminal locally. When a function key is pressed, data are transmitted to the computer and are available to the program which sent the formatted screen.

Block mode terminals typically have capabilities for defining protected and unprotected fields and sometimes allow a minimal set of character validations such as restricting a field to only allow digits. They do not provide JAM–like capabilities such as shifting, scrolling and provisions for post–field validation. It should therefore seem obvious that an application will behave slightly differently on a block mode terminal than on an interactive one. The goal of the block mode interface, however, is to minimize these differences and, to the greatest extent possible, allow applications to be created

that can operate in either mode without the need for the programmer to consider the differences. This is in keeping with the JAM philosophy of creating terminal–independent applications.

11.1.2
Authoring

Certain JAM utilities, like `modkey`, the Screen Editor, and the Data Dictionary Editor only work in interactive mode. Thus, they can only be used with interactive terminals or those that can be switched programmatically between block and interactive mode.

`jxform` is the JAM authoring utility. It allows the user to navigate through the screens in an application and to invoke the Screen and Data Dictionary Editors when appropriate. When used with block mode–only terminals, `jxform` does not permit entry into the aforementioned utilities. When used with hybrid terminals (i.e. those that can switch between block and interactive mode programmatically), `jxform` forces interactive mode before entering the utilities.

11.1.3
Selecting Block Mode

JAM operates with three types of terminals: interactive–only, block mode–only, and hybrid. Block mode can be used with either of the latter two.

By default, JAM operates in interactive mode regardless of the terminal type. To operate in block mode requires a block terminal driver to be linked with the system. (Block terminal drivers are described in detail later.) This alone, however, will not initiate block mode; two additional things must be done.

First there must be a call to `sm_blkinit`. This is generally done in the "main" routine of the application, `jmain.c`. If this call is absent, the application will be run in interactive mode. Also the additional code to support block mode will not be linked with the program. Thus programs not desiring block mode support are not penalized.

Second the correct block mode driver must be selected. This can be done in one of two ways.

If the application program author knows the correct driver he/she can install it by calling `sm_install`. This should be done before calling `sm_blkinit`. Typically the program will install a "hard–coded" driver, but it could instead key off of SMTERM, or some other environment variable, to find the correct one. In this case the application will run in block mode, independent of the end user's preference.

The second method for selecting the driver leaves the job to the end user. If `sm_blkinit` is called without previously installing a driver, the entry BLKDRIVER in the video file is examined. If it is absent, `sm_blkinit` fails and the application remains in interactive mode. If it is present the name given there is used to find the correct driver. This is done by a table lookup in a source routine (`blkdrvr.c`) that must be linked with the application. Naturally all possible choices of the driver must also be linked with the program. In this case the end user can override the application programers desire to use block mode.

The design allows for three scenarios: the programmer can prohibit block mode (no call to `sm_blkinit`), the programmer can force block mode (`sm_install` followed by `sm_blkinit`), or the programmer can permit block mode but allow the end user final say (`sm_blkinit` only).

Note that the application never calls `sm_blkdrvr`. The source code to that routine is given to customers to enable them to extend the capabilities of the second method.

11.1.4

Differences Between Block Mode And Interactive Mode

Although every attempt has been made to preserve the look and feel of applications operating in block mode, the following differences between block mode and interactive mode should be noted.

Screens

Screens work much as they do in interactive mode. The only noticable difference is that the cursor is not restricted to the active window as this is not possible in block mode. In keeping with the concepts of interactive mode, however, only the fields on the active window are unprotected.

Menus

In interactive mode, menus utilize a "bounce bar" to track the cursor. The bounce bar moves when cursor-positioning keys are pressed and when ASCII data are typed. Since block mode terminals do not return these keys, another approach must be taken. We supply two options:

In option 1, menu fields in block mode are unprotected, making it easy for an operator to tab to them. To make a selection, the operator positions to the appropriate field and

presses XMIT. Thus, selection is similar to interactive mode except there is no bounce bar and there is no provision for selecting by typing the first N characters of the menu choice.

If the operator inadvertently types over a menu field there are no adverse consequences as JAM will "remember" the contents and restore it at an appropriate time.

This approach works well since the same screens can be used for block and interactive mode operation. However, for those who do not wish to allow the operator to type over menu choice fields, option 2 may be chosen. With option 2, JAM creates an unprotected field to the left of each menu choice so the menu fields themselves can remain protected. The operator can tab to these new fields to make a selection, or type the first character of a menu field and press XMIT. The new fields to the left of the menu choices are created as long as there is room on the screen even if it means they would be placed in a border or a separate window. If there is no room on the screen because the menu field starts in position 1 or 2, the system reverts to option 1.

The above works well for traditional menus, but two–level (pull– down) menus pose a different problem in that the ONLY way to move horizontally in interactive mode is via the arrows (since TAB moves between the entries of the sub–menu). Thus, in block mode the following happens. When a pull–down menu is active, JAM unprotects all main menu fields except the one with which the pull–down is associated. Thus, the operator can either make a selection from the pull–down or tab to another main menu choice and press XMIT causing its sub–menu to be activated.

The two options for processing menus described above work equally well for pull–down menus.

Character Validation

The block mode interface takes advantage of the terminal's capabilities for character validation. However, for situations in which the specified validations go beyond what the terminal can handle, JAM will validate the character data during Screen Validation. The result will be something like this:

The operator enters alphabetic data in a digits–only field. When the XMIT key is pressed, all fields are validated in the normal fashion, left–to–right, top–to–bottom. Thus, the cursor will be positioned to the errant field and a message displayed.

Since programs do not rely on data being correct unless and until Screen Validation completes without error, this should pose no problem. The only consideration is that invalid character data can get into the screen buffer and LDB if the operator enters incorrect characters and then presses something like EXIT (this cannot happen in interactive mode because the invalid characters would not be allowed in the first place.

The only reason for mentioning this has to do with how punctuation characters in digits–only fields are handled. Let's say that a digits–only field got filled with slash ('/')

characters and this, in turn, got transferred to the screen buffer and hence to the LDB. On a subsequent attempt to enter data into the field, an attempt to merge the slashes with the entered data would be made. But since the field has ALL slash characters, there would be no room for the digits.

Thus, to eliminate the possibility of "punctuation character creep", when reading data from a digits–only field, JAM first strips out all punctuation characters from the field and then merges in the punctuation characters from the screen buffer.

Field Validation

Clearly, fields are not validated when TAB and RETURN are pressed as in interactive mode. Thus, like character validations, field validations will be deferred until Screen Validation. This should not be a problem since, even in interactive mode, the operator can usually bypass field validation by using the arrow keys to move from field to field. Therefore, programs should not rely on the data until Screen Validation passes without error in either mode.

One type of field validation is worth noting. Consider a field with an attached function which does a database lookup and displays information in another field. In interactive mode, this would usually be executed when the field is completed, so the user would see the result. Since this is not really a validation, deferring it until Screen Validation would not help because the data would never be seen by the operator. Therefore, if this type of feature is contemplated in a block mode environment, the database lookup should be attached to a function key rather than as an attached function.

Screen Validation

Screen validation works the same in interactive and block mode. The cursor will be positioned to the first field in error and a message will be displayed to the operator.

Right Justified Fields

Unless the block mode terminal supports this feature directly, the cursor will always be positioned to the left side of right justified fields when the cursor enters them.

Field Entry Function, Automatic Help, Status Text, etc.

These are disabled in block mode since JAM does not know when fields are entered.

Currency Fields

Currency edits are usually applied to fields as they are exited. In block mode, since this is not possible, currency formatting is done during screen validation. Care should be

taken with right justified currency formats since subsequent entry may be difficult for the reasons cited above in the section on right justified fields.

Shifting Fields

Normally fields shift when the left or right arrows are pressed with the cursor at the start or end of a shifting field or, in the case of unprotected fields, when the operator types off the edge of the field. Since arrows and data entry keys are not returned in block mode, this is not possible. To utilize shifting fields in block mode, use the logical keys: Shift Left and Shift Right. These shift the field by the shifting increment and work equally well in block and interactive mode.

An alternative is to use the Zoom feature if all shifting fields are limited to the width of the screen.

Scrolling Fields

This is similar to the situation with shifting fields. In block mode, one can define function keys as PAGE UP and PAGE DOWN, or use the Zoom feature.

Messages

Error messages are normally acknowledged by pressing the space bar, although the specific key used can vary depending on the setting of error message options. Also, options govern whether the key should be used as the next keystroke or discarded after the message is acknowledged. In block mode, ANY key that gets transmitted from the terminal will suffice to acknowledge messages, regardless of what key is defined for that purpose. Using or discarding the acknowledgement key apply equally to block mode and interactive mode.

With query messages, JAM normally expects a Y or N response. In block mode, JAM will create a field on the status line into which the Y or N response can be entered. This entry must be followed by the XMIT key for it to be accepted. On terminals that have a separate stauts line it is not possible to create such a field. In these cases, XMIT will be treated as a positive response; EXIT will be treated as a negative response.

Insert Mode

Insert mode will operate in whatever way the block mode terminal supports. However, since JAM never knows if insert mode is set or not in block mode, it will, for terminals in which this is a problem, reset insert mode before transmitting data to the terminal. This is so the new data will not be INSERTED into the terminal buffer, causing all other data to move around.

Non–Display Fields

If the block mode terminal supports this feature, it will be used.

System Calls

These operate as in interactive mode. However, before passing control to the OS, JAM sets the terminal to the mode (block or interactive) expected by the OS, and resets it upon return from the system call. The JAM routines `sm_leave` and `sm_return` do the same.

Zoom

With the exception of the limitations expressed in the sections on shifting and scrolling, Zoom works as in interactive mode.

Help and Item Selection

With the exception of the limitations expressed in the sections on shifting, scrolling, field entry and menu processing, these functions work as in interactive mode.

Groups

Radio buttons and check lists behave similar to menus as described above.

11.2 WRITING A BLOCK MODE DRIVER

11.2.1 Installation

There are two parts to the installation process. These were discussed in greater detail above.

First a block terminal driver must be installed. This driver performs the low level communication between JAM and the terminal.

Next the application program must initiate block mode by making the appropriate subroutine call. The application program can also switch to interactive mode by means of

a call. The assumption is that the default is interactive mode, thus a call to set block mode is needed even if that is the normal mode of the operating system. The application program can also set some operating parameters by means of a subroutine call.

We discuss these steps in reverse order: application program support, and block mode terminal driver.

11.2.2

Application Program Support

JAM programs assume that the terminal is in interactive mode. Explicit calls are needed to switch from interactive to block and vice versa. To turn on block mode, the program should call `sm_blkinit`. To turn off block mode (and turn on interactive mode) the program calls `sm_blkreset`. The Screen Editor and the key mapping utility (`modkey`) also require interactive mode. The authoring utility (`jxform`) can be made to work in block mode, switching to interactive mode when the Screen Editor is invoked. This can be done by inserting the appropriate calls in `jxmain.c` (provided) and relinking `jxform`.

The routine `sm_option` can be used to set some user–preference items.

11.2.3

Block Terminal Driver

There is a single entry point to the block mode terminal driver. A request code is passed by JAM to specify the action required of the driver.

The request codes are listed in `smblock.h`. They can be organized into groups. There are request codes for initialization and resetting the terminal. Next there are requests for locking and unlocking visual (e.g. underlined) and logical (e.g. protected) attributes. The driver will generally implement these by sending simple sequences to the terminal.

The next requests are used for setting the logical attributes on the various fields. This action is highly terminal specific, and is the most complicated chore of the driver.

Next come the requests for opening and closing the keyboard, and retrieving characters from it. Here the driver may process the special handshaking required by the terminal. In addition the driver may have to determine (and save) the position of the cursor on the terminal, as this information often comes as part of the data stream from the terminal when a key it hit.

The next set of requests are used to obtain data from the terminal. Parsing the data stream is terminal specific, however it is usually quite simple.

Finally JAM asks the driver for a cursor position report.

The driver is installed by calling `sm_install`. The driver can be written in any computer language supported by that installation of JAM. The discussion below assumes that the driver is written in C. The definition of `struct fnc_data` is in `smdefs.h`:

```
struct fnc_data blkdrvr_fnc =
{
    (char *)0, blkdriver,  0, 0, 0, 0
};
int numentries = 1;

sm_install (BLKDRVR_FUNC, blkdrvr, &numentries);
```

The driver is called with one parameter. It is the address of an instance of `struct blk_data_block` (defined in `smblock.h`). Other computer language implementations may have different parameters. This structure is static in `block.c`; the same address will be passed on every call. The driver is free to use this information if it desires. (It could save the address on the initialization call and then ignore the parameter on other calls.)

The structure is defined below.

```
struct blk_data_block
{
    int type;       /* type of request to driver  */
    int value;      /* general purpose int value  */
    int line;       /* line or number of lines*/
    int colm;       /* colm or number of colms*/
    struct field_data *data;/* field info for set log att   */
    char *buf;      /* work buffer, or return val */
};
```

The interpretation of the return code of the driver is specific to the request being made. The "usual" return code is 0. "Type" specifes the type of request, for example initialize or reset. "Value" is a general purpose value passed to the driver or, on rare occasions, returned from the driver.

"Line" and "colm" are also used for different purposes. On some calls they are set to the screen size. On others they represent the start line and column of a screen area. "Data" is only used for setting logical attributes on a field.

The last member, buf, is set to a static work buffer of 256 bytes. This buffer may be used by the driver for any purpose. On some calls the value in this member is returned by the driver. The data can either be put into the buffer passed, or a new pointer can be loaded into this member. (Thus this member is reloaded with the address of the work buffer before every call.)

11.2.4

Driver Request Types

The individual request types are listed here and detailed on individual manual pages below.

Request Type	*Description*	*Page*
BLK_INIT	initialization call and set block mode	141
BLK_RESET	reset	142
BLK_BLOCK	set block without initialization	143
BLK_CHAR	set character mode independent of os mode	144
BLK_LOCK	lock visual and logical attributes	145
BLK_UNLOCK	unlock visual and logical attributes	146
BLK_LA_START	start of log attribute calls	147
BLK_LA_UNPROT	set logical attribute	149
BLK_LA_PROT	set protected attribute	150
BLK_LA_END	end of logical attribute calls	151
BLK_K_OPEN	open keyboard for data entry	152
BLK_K_GETCHAR	get a character from the keyboard	153
BLK_K_CLOSE	close keyboard (lock it)	155
BLK_D_START	start getting data from terminal	157
BLK_D_UNPROT	unprotected field	159
BLK_D_PROT	protected field	160
BLK_D_END	end of getting data from terminal	161
BLK_CPR	report the cursor position to JAM	162

BLK_INIT
initialize terminal for block mode

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_INIT
blk_data->value = ?
blk_data->line  = number of lines on screen
blk_data->colm  = number of columns on screen
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This call is made to do initialization and put the terminal into block mode. If the terminal cannot be changed, –1 should be returned.

The terminal should be put into block mode, data transmission protocols should be established, and the editing keys made local. If possible, the keyboard should be locked. JAM will call the driver to unlock the visual and logical attributes, thus that task need not be done by the present routine.

RETURNS

0 if the terminal can be put into block mode.
–1 if the terminal cannot be put into block mode.

BLK_RESET
reset terminal to operating system mode

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_RESET
blk_data->value = ?
blk_data->line  = number of lines on screen
blk_data->colm  = number of columns on screen
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This call is made to put the terminal out of block mode and into the mode desired by the operating system. If the terminal cannot be changed, the driver returns –1.

If the terminal is to be put into interactive mode, the editing keys should be made "transmit". JAM will call the driver to unlock the visual and logical attributes, but will not make any call to unlock the keyboard.

RETURNS

0 if the terminal can be put into interactive mode.
–1 if the terminal cannot be put into interactive mode.

BLK_BLOCK
set terminal to block mode without initialization

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_BLOCK
blk_data->value = ?
blk_data->line  = number of lines on screen
blk_data->colm  = number of columns on screen
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This call is similar to BLK_INIT except it is only used to put the terminal back into block mode after it has been switched to character mode during menu_proc or error_reset. If the terminal cannot be changed, –1 should be returned.

The terminal should be put into block mode, data transmission protocols should be established, and the editing keys made local. If possible, the keyboard should be locked. JAM will call the driver to unlock the visual and logical attributes, thus that task need not be done by the present routine.

RETURNS

0 if the terminal can be put into block mode.
–1 if the terminal cannot be put into block mode.

BLK_CHAR
set terminal to character mode

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_CHAR
blk_data->value = ?
blk_data->line  = number of lines on screen
blk_data->colm  = number of columns on screen
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This call is made to put the terminal into character mode temporarily to handle interactive menu_proc or error_reset. If the terminal cannot be changed, the driver returns –1.

The terminal should be put into interactive mode. The editing keys should be made "transmit".

RETURNS

0 if the terminal can be put into interactive mode.
–1 if the terminal cannot be put into interactive mode.

BLK_LOCK
lock visual and logical attributes

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_LOCK
blk_data->value = ?
blk_data->line  = number of lines on screen
blk_data->colm  = number of columns on screen
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This call is made to lock the current visual and logical attributes to the screen position. It is called immediately before opening the keyboard.

Often the terminal manual calls locking "asserting the logical attributes" or setting "format mode". All settings should be made so the operator can only enter data into unprotected fields. On some terminals, "erasure" mode should be set so display characters cannot be erased.

RETURNS

the return code is ignored.

BLK_UNLOCK
unlock visual and logical attributes

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_UNLOCK
blk_data->value = ?
blk_data->line  = number of lines on screen
blk_data->colm  = number of columns on screen
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This call is made to unlock the current visual and logical attributes to the screen position. It is called before clearing the screen or window, when bringing up a form or window, and before sending any text or attributes to the terminal.

The logical attributes should not be asserted, format mode turned off, erasure mode should be turned off. All settings should be such that any text or attributes on the screen may be changed by JAM.

RETURNS

the return code is ignored.

BLK_LA_START
start of logical attribute calls

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_LA_START
blk_data->value = 0 if no unprotected fields, 1 otherwise
blk_data->line  = number of lines on screen
blk_data->colm  = number of columns on screen
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This call is made before setting logical attributes (e.g. protect or numeric). At the time of this call, the screen is already set up with the correct visual attributes.

This is the "initialization" call. The return code specifies how subsequent calls should be made. The mnemonics in `smblock.h` are listed below:

```
BLK_BACKWARD
BLK_DISPLAY
BLK_ATT_CHANGE
BLK_LINE
```

The default (0) is that there be 1 call for each unprotected field. The order is from left to right, top to bottom (field number order). If this option is chosen, the driver should protect the entire screen on the initialization call. Then, on each subsequent call a single field will be made unprotected.

For terminals that cannot protect the entire screen, a sub– seqent call will have to be made for each protected and unprotected area on the screen. For such terminals, the driver does nothing on initialization but returns BLK_DISPLAY.

Normally a display area is the area between two fields. However some terminal drivers may need finer division. If the BLK_ATT_CHANGE bit is set, a display area is defined to be the largest area of a single (visual) attribute. If the BLK_LINE bit is set, no display area will extend across more than one line.

Some terminals are more efficient if the order is reversed (last field first). This is specified by the low bit, mnemonic BLK_BACKWARD.

Some terminals define one visual attribute for protected areas (e.g. dim), all others for unprotected areas. In this case, the initialization should do nothing. It should request

BLK_DISPLAY | BLK_ATT_CHANGE. It must then change the visual attributes to match the protection requirements.

RETURNS

-1 if the driver cannot set any logical attributes.
the kind of requests the driver needs to set logical attributes (see above).

BLK_LA_UNPROT
set logical attribute on unprotected fields

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_LA_UNPROT
blk_data->value = length of field
blk_data->line  = start line of field
blk_data->colm  = start column of field
blk_data->data  = field_data structure pointer
blk_data->buf   = work_buffer
```

DESCRIPTION

This call is made to set the logical attributes of an unprotected field. The "data" member of the structure contains a pointer to the field_data structure for that field. Thus all the information about the field is available to the driver. Whatever logical attributes supported by the terminal can be set.

RETURNS

the return code is ignored.

BLK_LA_PROT
set protected attribute

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_LA_PROT
blk_data->value = length of area
blk_data->line  = start line of area
blk_data->colm  = start column of area
blk_data->data  = 0 if display area, else field_data struct pointer
blk_data->buf   = work_buffer
```

DESCRIPTION

The call is made only if requested in the BLK_LA_START return code. The member "data" of the structure is 0 if this call represents a display area, or is the field_data structure pointer for a protected field.

The area in question can span lines (unless BLK_LINE was set on the initialization call). It need not have a constant attribute (unless BLK_ATT_CHANGE was set.) Attribute (visual) information can be obtained from `sm_attrib`.

RETURNS

the return code is ignored.

BLK_LA_END
end of logical attribute calls

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_LA_END
blk_data->value = ?
blk_data->line  = number of lines on screen
blk_data->colm  = number of columns on screen
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This is the termination call. Most drivers will do nothing. The call is present to allow a "clean–up", for example to free memory no longer needed.

This call will not be made if BLK_LA_START returns –1.

RETURNS

the return code is ignored.

BLK_K_OPEN
open keyboard for operator input

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_K_OPEN
blk_data->value = ?
blk_data->line  = ?
blk_data->colm  = ?
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This is the initialization call. The driver should open the keyboard and allow operator entry. This call will normally not wait until a response is available. It's duty is to simply allow the response.

RETURNS

the return code is ignored.

BLK_K_GETCHAR
get characters from the keyboard

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_K_GETCHAR
blk_data->value = ?
blk_data->line  = ?
blk_data->colm  = ?
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This request is made to obtain a character(s) from the keyboard. Normally, a block-mode terminal will transmit data when a function key or a "send" key is pressed. In the former case, this routine will send back the characters generated on behalf of the key-stroke; in the latter case, the routine will generally send back an indication that the XMIT key was pressed.

This driver entry point should block waiting for a response from the terminal. When the response comes, it is up to the driver to perform any handshaking required. If necessary, the driver may ask the terminal for a cursor position report. The response obtained should not contain any handshaking characters, nor any block termination character. Thus the response should be the same as if the terminal were in interactive mode.

The driver returns the character(s) in the member "buf". Multiple characters may be returned, "retcode" should be set to the number of characters returned. Upon receipt of the blk_data structure, "buf" is set to a work space of 256 bytes. The driver may simply put the characters into this buffer. However, if desired, a different buffer may be used and the address passed back in this structure member. The sequence of keys in buf need not be null–terminated.

This entry point may be called multiple times. For example, the F1 key may be designated as a "shift" or lead–in key. Thus F1 by itself is not a logical key. In this case the driver will be called and return F1. The driver will be called again to obtain the next key.

If the driver wishes to return a logical key rather than the raw characters, it should set the "value" member of the structure to the logical key value and return 0.

For example, if the user presses the key that sends data to the computer (SEND or ENTER or the like), the driver will likely save the first character for later use in the blkdata entry point, and send a XMIT back. Thus "value" would be set to XMIT and retcode to 0.

If, for some reason, the driver wishes to send nothing back it should set value to 0 and return 0.

RETURNS

retcode is the count of keys buffered in "buf".
if retcode is 0, a logical key value (e.g. XMIT) should be returned in "value".

BLK_K_CLOSE
close keyboard (lock it)

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_K_CLOSE
blk_data->value = see below
blk_data->line  = ?
blk_data->colm  = ?
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

After getkey has sucessfully translated the keystrokes into a logical value, the driver is called again with request BLK_K_CLOSE and value set to the translated logical value.

The driver can perform further translation if required by replacing value by the new logical key. (This duplicates the function of the keychange hook.)

This entry point should close the keyboard for operator entry.

Consider the case where the terminal participates in handshaking.

In the first call (BLK_K_OPEN), the keyboard is opened.

In the next call (BLK_K_GETCHAR), the "enable" handshake is sent to the terminal. The driver then waits for a response from the terminal. When it (the terminal's request to send) is received, the driver asks the terminal for a cursor position report and saves the answer. It then homes the cursor (in case the user hit the "send" key) and sends the "clear to send" handshake. Now the terminal responds.

If the first character is ESC, this is a function key and the driver buffers the keys in the work buffer until the block terminator is found. This terminator is discarded.

If the first character is not ESC, the user hit the send data key. This character is saved for the getdata entry point and the value XMIT is returned to JAM.

In the final call (BLK_K_CLOSE), the driver simply closes the keyboard.

Consider the case in which the terminal does not use handshaking.

The first call (BLK_K_OPEN) simply opens the keyboard.

The next call (BLK_K_GETCHAR) waits for a response. That response will always start with a cursor position report which the driver parses and saves. If the next key is ESC, the entire sequence up to the block terminator is returned to JAM. If the key is not ESC, it is saved and XMIT is returned to JAM.

In the final call (BLK_K_CLOSE), the driver closes the keyboard.

RETURNS

the return code is ignored.

BLK_D_START
start getting data from terminal

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_D_START
blk_data->value = 1 if there are unprotected fields, 0 otherwise
blk_data->line  = number of lines on screen
blk_data->colm  = number of columns on screen
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This call functions similarly to the set logical attributes call. Its purpose is to update the logical screen buffer with the data entered by the keyboard operator.

This call is always made immediately after the BLK_K_CLOSE, and at no other time.

This is the initialization call. "Value" is set to tell whether or not there are unprotected fields on the form. Typically the driver will use this information to decide whether or not to poll the terminal for data. Indeed, if there are no fields, the driver will probably return –1 to JAM. In this case no further calls to the BLK_D_ series will be made.

The return code specifies how subsequent calls should be made. Mnemonics are in `smblock.h`.

The default (0) is that there be 1 call for each unprotected field. The order is from left to right, top to bottom (field number order). The low bit, BLK_BACKWARD can be used to reverse this order.

Some terminals send data for protected fields as well as unprotected fields. If the bit BLK_DISPLAY is set, calls will be generated for display areas as well as unprotected and unprotected fields.

Normally a display area is the area between two fields. However some terminal drivers may need finer division. If the BLK_ATT_CHANGE bit is set, a display area is defined to be the largest area of a single (visual) attribute. If the BLK_LINE bit is set, no display area will extend across more than one line.

If necessary, the driver should request that the terminal send data. (If value is 0 this step may be unnecessary.)

RETURNS

–1 if the driver need not return any data (for example if the form has no unprotected fields).

the kind of requests the driver needs to return data to JAM (see above).

BLK_D_UNPROT
get data from an unprotected field

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_D_UNPTROT
blk_data->value = length of field/data
blk_data->line  = start line of field
blk_data->colm  = start column of field
blk_data->data  = field_data structure pointer
blk_data->buf   = work_buffer
```

DESCRIPTION

The data for the field should be placed in "buf", either in the work buffer supplied or in a driver supplied buffer. "Value" should be set to the length of the data in that buffer. Normally this is the same as the length of the field (thus "value" need not be changed). Some terminals do not send trailing blanks. The driver could either reset "value" to the corret amount or could pad the data in "buf".

If the terminal reports that the field has not been modified, "value" should be set to 0. This signifies to JAM that the current value of the field should not be changed.

RETURNS

the return code is ignored.

BLK_D_PROT
get data from a protected area

SYNOPSIS

```
retcode = blkdriver (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_D_PROT
blk_data->value = length of area
blk_data->line  = start line of area
blk_data->colm  = start column of area
blk_data->data  = 0 for display areas, else field_data struct pointer
blk_data->buf   = work_buffer
```

DESCRIPTION

This call allows the driver to pass over data coming from the terminal that represents a protected area. No data is transfered to JAM.

RETURNS

the return code is ignored.

BLK_D_END
end of getting data from terminal

SYNOPSIS

```
retcode = blkdata (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_D_END
blk_data->value = ?
blk_data->line  = ?
blk_data->colm  = ?
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This is the termination call. Most drivers will do nothing. The call is present to allow a "clean–up", for example to free memory no longer needed. "Line" and "column" are set to the screen size.

Sometimes the driver will have to read (and discard) a final block terminator.

If BLK_D_START returned –1, this call will not be made.

RETURNS

the return code is ignored.

BLK_CPR
report cursor position to **JAM**

SYNOPSIS

```
retcode = blkdata (blk_data);
int retcode;
struct blk_data_block *blk_data;

blk_data->type  = BLK_CPR
blk_data->value = ?
blk_data->line  = cursor line returned to JAM
blk_data->colm  = cursor column returned to JAM
blk_data->data  = ?
blk_data->buf   = work_buffer
```

DESCRIPTION

This driver routine must determine and return to JAM the current cursor position. This is often the value saved much earlier in the process. However it could be obtained by polling the terminal.

The current line and column should be returned to JAM in the members "line" and "colm" of the structure.

Since JAM depends highly on knowing the cursor position the driver must support this entry point.

RETURNS

the return code is ignored.

11.2.5

Driver Support Routines

Most driver routines are called from within the "get character" routine of JAM. Thus the driver often should not use "high level" JAM functions. Guidelines for specific entry points are listed below.

For output to the terminal, the driver should use `sm_puchr` (single character – like putchar) or `sm_pustr` (string – like fputs).

To ensure that all delayed write buffers are written to the screen, `sm_flush` may be called. This is done by JAM before calling LA_START. If any of the LA_ routines alter the physical display (e.g. `sm_chg_attr` was called), LA_END should call `sm_flush`.

To position the cursor the routine `sm_tcursor` (line, colm) should be used. If the cursor is positioned in any other way JAM will not be aware of it. In this case the global integers `sm_tline` and `sm_tcolm` should be either set correctly or set to –1 (to indicate that cursor position is unknown).

Similarly attributes should be changed by calling `sm_chg_attr` or `sm_do_region`. If any visual attributes are changed by the driver, `sm_flush` should be called (usually in BLK_LA_END) to force out the "delayed write".

The display data can be found in `sm_screen`, attributes in `sm_attrib`. Note that these are the "logical" values. They have been sent to the terminal just before BLK_LA_START.

On area attribute terminals there is an additional buffer, `sm_exattr`. This buffer is a character array paralleling the screen. The declaration is in `smblock.h`:

```
char NEAR * NEAR * NEAR sm_exattr;
```

It is arranged so that it may be accessed as if it were a 2 dimensional array (`sm_exattr[line][colm]`). The high bit is set if that position contains an area attribute. The next highest bit is used for terminals that enforce a limit on the number of attributes on a given line.

The low 4 bits are available for use by the driver. Typically the driver will set these to indicate that a start field or end field attribute exists at the given position.

`sm_exattr` is cleared by clear window and clear to end of line.

For attributes two globals exist: `sm_lmask` and `sm_amask`. They contain a bit mask for those attributes that are supported by the terminal.

BLK_INIT and BLK_RESET are called on the "application" level. All JAM routines are available. The global buffer `sm_term` contains the environment entry SMTERM

or TERM, and may be used to set up conditional code in the driver, based on terminal type.

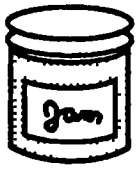
BLK_LOCK and BLK_UNLOCK are called while bringing up a form or window and before any writing to the screen. It is not safe to assume that JAM global variables are correctly set at this point. Only `sm_puchr`, `sm_pustr` and `sm_tcursor` should be used.

BLK_LA_??? are called just before opening the keyboard. All JAM routines that do not open a window or form or open the keyboard are available.

BLK_K_??? are called to open the keyboard. All JAM routines that do not open a window or form, or open the keyboard are available. If any change is made to the display or attributes, `sm_flush` should be called.

BLK_D_??? are called after opening the keyboard. All JAM routines that do not open a window or form, or open the keyboard are available.

BLK_CPR is called last in sequence.



Chapter 12
Library Function Overview

In this chapter, we summarize the JAM library functions and list them in categories. All JAM library function names begin with the prefix `sm_`. However, in the Function Reference Chapter and in this chapter, the functions are listed without prefix for clarity.

In addition to stripping off the prefix in the listings that follow, groups of closely related variant functions are listed under a single root name. The functions `sm_r_form`, `sm_d_form`, and `sm_l_form`, for example, are all grouped under the heading `form`. In a few cases, functions may be listed under a name that is not a portion of the the function name but is suggestive of the utility of the function. For example, the function `sm_r_at_cur`, which displays a window at the cursor position, is listed under the root name `window`, along with `sm_r_window` (which displays a window at a fixed location) and a number of other window display routines. The calling syntax of each function is found in the SYNOPSIS section of the function listing in the Function Reference Chapter.

Most JAM library routines fall into one of the following categories:

- Initialization/Reset
- Screen and Viewport Control
- Keyboard and Display I/O
- Field/Array Data Access
- Field/Array Characteristic Access
- Group Access
- Local Data Block Access
- Cursor Control
- Message Display

- Scrolling and Shifting
- Mass Storage and Retrieval
- Validation
- Global Data and Changing JAM's Behavior
- Soft Keys and Keysets
- JAM Executive Control
- Block Mode Control
- Miscellaneous

The following sections summarize the functions that fall into these categories. Some listings are found in more than one category.

12.1
INITIALIZATION/RESET

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

`cancel`	reset the display and exit
`dicname`	set data dictionary name
`do_uinstalls`	install an application's hook functions
`ininames`	record names of initial data files for local data block
`initcrt`	initialize the display and JAM data structures
`keyinit`	initialize key translation table
`ldb_init`	initialize (or reinitialize) the local data block
`leave`	prepare to leave a JAM application temporarily
`msgread`	read message file into memory
`resetcrt`	reset the terminal to operating system default state
`return`	prepare for return to JAM application
`vinit`	initialize video translation tables

12.2
SCREEN AND VIEWPORT CONTROL

The following routines are used to control viewports, the display of screens, and the form and window stacks.

`close_window`	close current window
`form`	display a screen as a form
`hlp_by_name`	display help window
`issv`	determine if a screen in the saved list
`jclose`	close current window or form under JAM Executive control
`jform`	display a screen as a form under JAM control
`jwindow`	display a window at a given position under JAM control
`mwindow`	display a status message in a window
`shrink_to_fit`	remove trailing empty array elements and shrink screen
`sibling`	define the current window as being or not being a sibling window
`submenu_close`	close the current submenu
`svscreen`	register a list of screens on the save list
`unsvscreen`	remove screens from the save list
`viewport`	modify viewport size and offset
`wcount`	obtain number of currently open windows
`wdeselect`	restore the formerly active window
`window`	display a window at a given position
`winsize`	allow end–user to interactively move and resize a window
`wselect`	activate a window
`wrotate`	rotate the display of sibling windows

12.3
DISPLAY TERMINAL I/O

The following routines provide the interface to JAM terminal I/O.

`bel`	beep!
`bkrect`	set background color of rectangle

`do_region`	rewrite part or all of a screen line
`flush`	flush delayed writes to the display
`getkey`	get logical value of the key hit
`input`	open the keyboard for data entry and menu selection
`keyfilter`	control keystroke record/playback filtering
`keyhit`	test whether a key has been typed ahead
`keylabel`	get the printable name of a logical key
`keyoption`	set cursor control key options
`m_flush`	flush the message line
`rescreen`	refresh the data displayed on the screen
`resize`	dynamically change the size of the display
`ungetkey`	push back a translated key on the input

12.4
FIELD/ARRAY DATA ACCESS

The following routines access the data in fields and arrays. Most routines in this section have a number of variants that perform the same task but reference the field to be accessed differently. In these cases, the calling syntax of the *major* variant is listed under the SYNOPSIS section of the listing in the Function Reference Chapter. All other variants are listed under the VARIANTS section. There are also listings for each prefix that explain how they work (for example there is a reference listing for `i_`).

Most field access routines have five variants, although some have fewer. The five possible variants are shown in the table below:

Variants of Functions That Access Fields		
Prefix	*Example*	*Description*
`sm_`	`sm_intval(fieldnum);`	Access a field via field number.
`sm_n_`	`sm_n_intval(fieldname);`	Access a field (or an entire array) via field name. Access the LDB if there is no field on the screen.

Prefix	*Example*	*Description*
sm_i_	sm_i_intval(fieldname, occurrence);	Access an occurrence via field name and occurrence number. Access the LDB if there is no field on the screen.
sm_o_	sm_o_intval(fieldnum, occurrence);	Access an occurrence via field number and occurrence number.
sm_e_	sm_e_intval(fieldname, element);	Access an element via field name and element number.

amt_format	write data to a field, applying currency editing
calc	execute a math edit style expression
cl_unprot	clear all unprotected fields
clear_array	clear all data in an array
dblval	get the value of a field as a real number
dlength	get the length of a field's contents
doccur	delete occurrences
dtofield	write a real number to a field
fptr	get the content of a field
getfield	copy the contents of a field
gwrap	get the contents of a wordwrap array
intval	get the integer value of a field
ioccur	insert blank occurrences into an array
is_no	test field for no
is_yes	test field for yes
itofield	write an integer value to a field
lngval	get the long integer value of a field
ltofield	place a long integer in a field
null	test if field is null
putfield	put a string into a field
pwrap	put text to a wordwrap field
strip_amt_ptr	strip amount editing characters from a string

12.5 FIELD/ARRAY ATTRIBUTE ACCESS

The following routines access information about fields and arrays. Like the routines in the previous section on field and array data access, each of these routines generally have five distinct variants. See the discussion in the introduction to the previous section for more information on variants of JAM library functions that access fields.

`base_fldno`	get the field number of the first element of an array
`bitop`	manipulate validation and data editing bits
`chg_attr`	change the display attribute of a field
`cl_all_mdts`	clear all MDT bits
`dlength`	get the length of a field's contents
`edit_ptr`	get special edit string
`finquire`	obtain information about a field
`fldno`	get the field number of an array element or occurrence
`ftog`	convert field references to group references
`ftype`	get the data type and precision of a field
`gtof`	convert a group name and index into a field number and occurrence
`length`	get the maximum length of a field
`max_occur`	get the maximum number of occurrences
`name`	obtain field name given field number
`num_occurs`	find the highest numbered occurrence containing data
`protect`	protect an array
`sc_max`	alter the maximum number of items allowed in a scrollable array
`size_of_array`	get the number of elements
`tst_all_mdts`	find first modified occurrence in the screen

12.6
GROUP ACCESS

The following routines access groups, that is, radio buttons and check lists. Groups are made up of fields that have attributes and data in them, but groups in and of themselves are implemented as phantom fields which take up no screen real estate. The value of a group indicates the set of selected consituent fields, although it is not recommended that that value ever be accessed or modified directly with any of the field access routines discussed in the preceding sections.

The routines that follow are those that are recommended for accessing groups:

`deselect`	deselect a checklist occurrence
`ftog`	convert field references to group references
`gp_inquire`	obtain information about a group
`gtof`	convert a group name and index into a field number and occurrence
`isselected`	determine whether a radio button or checklist occurrence has been selected
`select`	select a checklist or radio button occurrence

12.7
LOCAL DATA BLOCK ACCESS

The following routines access the Local Data Block, or LDB. Note that any of the field data access routines that reference fields by name or name and occurrence number (eg `sm_n` and `sm_i_` variants) will access the LDB if the named field does not exist on the active screen.

`allget`	load screen from the LDB
`dicname`	set data dictionary name
`dd_able`	turn LDB write–through on or off
`ininames`	record names of initial data files for local data block
`lclear`	erase LDB entries of one scope
`ldb_init`	initialize (or reinitialize) the local data block
`lreset`	reinitialize LDB entries of one scope
`lstore`	copy everything from screen to LDB

12.8
CURSOR CONTROL

The following routines control the positioning and display of the cursor on the active screen.

`ascroll`	scroll to a given occurrence
`backtab`	backtab to the start of the last unprotected field
`c_off`	turn the cursor off
`c_on`	turn the cursor on
`c_vis`	turn cursor position display on or off
`disp_off`	get displacement of cursor from start of field
`getcurno`	get current field number
`gofield`	move the cursor into a field
`home`	home the cursor
`last`	position the cursor in the last field
`nl`	position cursor to the first unprotected field beyond the current line
`occur_no`	get the current occurrence number
`off_gofield`	move the cursor into a field, offset from the left
`rscroll`	scroll an array
`sh_off`	determine the cursor location relative to the start of a shifting field
`tab`	move the cursor to the next unprotected field

12.9
MESSAGE DISPLAY

The following routines are intended for the access and display of runtime application messages.

`d_msg_line`	display a message on the status line
`emsg`	display an error message and reset the message line, without turning on the cursor
`err_reset`	display an error message and reset the status line

m_flush	flush the message line
msg	display a message at a given column on the status line
msg_get	find a message given its number
msgfind	find a message given its number
msgread	read message file into memory
mwindow	display a status message in a window
query_msg	display a question, and return a yes or no answer
qui_msg	display a message preceded by a constant tag, and reset the message line
quiet_err	display error message preceded by a constant tag, and reset the status line
setbkstat	set background text for status line
setstatus	turn alternating background status message on or off

12.10
SCROLLING AND SHIFTING

The following routines provide access to shifting and scrolling fields and arrays.

achg	change the display attribute of an occurrence within a scrolling array
ascroll	scroll to a given occurrence
doccur	delete occurrences
ioccur	insert blank occurrences into an array
max_occur	get the maximum number of occurrences
num_occurs	find the highest numbered occurrence containing data
oshift	shift a field by a given amount
rscroll	scroll an array
sc_max	alter the maximum number of items allowed in a scrollable array
sh_off	determine the cursor location relative to the start of a shifting field
t_scroll	test whether an array can scroll
t_shift	test whether field can shift
tst_all_mdts	find first modified occurrence

12.11
MASS STORAGE AND RETRIEVAL

The following routines move data to or from sets of fields in the screen or LDB.

`rd_part`	read part of a data structure to the current screen
`rdstruct`	read data from a structure to the screen
`restore_data`	restore previously saved data to the screen
`rrecord`	read data from a structure to a data dictionary record
`rs_data`	restore saved data to some of the screen
`save_data`	save screen contents
`sv_data`	save partial screen contents
`sv_free`	free a save–data buffer
`wrecord`	write data from a data dictionary record to a structure
`wrt_part`	write part of the screen to a structure
`wrtstruct`	write data from the screen to a structure

12.12
VALIDATION

The following routines provide an application interface to the field and group validation processes.

`bitop`	manipulate validation and data editing bits
`ckdigit`	validate check digit
`fval`	force field validation
`gval`	force group validation
`novalbit`	forcibly invalidate a field
`s_val`	validate the current screen

12.13
GLOBAL DATA AND CHANGING JAM'S BEHAVIOR

The following routines grant access to global data and provide a way to manipulate certain aspects of JAM and Screen Manager behavior.

`dd_able`	turn LDB write-through on or off
`finquire`	obtain information about a field
`gp_inquire`	obtain information about a group
`inquire`	obtain value of a global integer variable
`install`	install application functions
`isabort`	test and set the abort control flag
`iset`	change value of global integer variable
`keyfilter`	control keystroke record/playback filtering
`keyoption`	set cursor control key options
`msgread`	read message file into memory
`option`	set a Screen Manager option
`pinquire`	obtain value of a global string
`pset`	modify value of global strings
`resize`	dynamically change the size of the display
`soption`	set a string option

12.14
SOFT KEYS AND KEYSETS

The following routines provide an application interface to JAM's soft key support.

`c_keyset`	close a keyset
`keyset`	open a keyset
`kscscope`	query current keyset scope
`ksinq`	inquire about key set information
`kslabel`	set a soft key label and attribute (`sm_skset` is preferred)
`ksoff`	turn off key labels
`kson`	turn on key labels
`skinq`	obtain soft key information by position
`skmark`	mark or unmark a softkey label by position
`skset`	set characteristics of a soft key by position
`skvinq`	obtain soft key information by value
`skvmark`	mark a soft key by value
`skvset`	set characteristics of a soft key by value

12.15
JAM EXECUTIVE CONTROL

The following routines, available only to applications using the JAM Executive, provide JAM Executive services.

`getjctrl`	get control string associated with a key
`jclose`	close current window or form under JAM Executive control
`jform`	display a screen as a form under JAM control
`jtop`	start the JAM Executive
`jwindow`	display a window at a given position under JAM control
`putjctrl`	associate a control string with a key

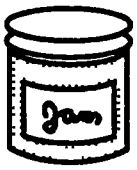
12.16
BLOCK MODE CONTROL

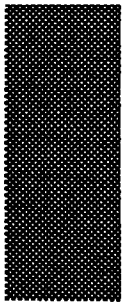
The following routines are used in applications requiring block mode support.

`blkdrvr`	install block mode driver
`blkinit`	initialize (and turn on) block mode terminal
`blkreset`	reset (and turn off) block mode terminal

12.17
MISCELLANEOUS

`fi_open`	find a file and open it in binary read only mode
`fi_path`	return the full path name of a file
`formlist`	update list of memory–resident files
`jplcall`	execute a JPL procedure
`jplload`	execute the JPL load command
`jplpublic`	execute the JPL public command
`jplunload`	execute the JPL unload command
`l_close`	close a library
`l_open`	open a library
`rmformlist`	empty the memory–resident form list
`sdtime`	get formatted system date and time
`udtime`	format user–supplied date and time





Chapter 13
Function Reference

All JAM function names begin with the prefix `sm_`. In the Function Reference Chapter functions are listed without the prefix and, in a few cases, under a name that is not a portion of the function name — but that is suggestive of the utility of the function. For example, the function `sm_r_at_cur`, which displays a window at a specified position, is found under the listing name `window`, along with the function `sm_r_window`. In these cases, the calling syntax of each function is listed under the SYNOPSIS section of the listing.

For each entry, you will find several sections:

- A synopsis similar to a C function declaration, giving the types of the arguments and return value.
- A description of the function's arguments, prerequisites, results, and side-effects.
- The function's return values, if any, and their meanings.
- A list of variants.
- A list of functions that perform related tasks.
- An example illustrating the function's use.

A routine that calls JAM functions should include the file `smdefs.h`. If another file should be included, then it is referenced in the synopsis section.

To view functions by category, refer to the Library Function Overview (Chapter 12) To view a complete list of functions alphabetically by the actual function name (including the `sm_` prefix), see the Library Function Index (Chapter 14).

achg

change the display attribute of an occurrence within a scrolling array

SYNOPSIS

```
int sm_o_achg(field_number, occurrence, display_attribute)
int field_number;
int occurrence;
int display_attribute;
```

DESCRIPTION

NOTE: This function has only two variants, `sm_o_achg` and `sm_i_achg`. There is no `sm_achg`.

This function changes the display attribute of an occurrence within a scrollable array. If the occurrence is onscreen, the attribute with which the occurrence is currently displayed is changed as well. When the occurrence is scrolled to another position within the array the new attribute moves with the occurrence. Use `sm_chg_attr` if you want all of the occurrences within the array to scroll through an attribute so that their appearance is determined by their onscreen positions.

Possible values for the argument `display_attribute` are defined in the header file `smattrib.h`, as shown in the table below:

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Highlights		Background Highlights	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Colors		Background Colors	
BLACK	0000	B_BLACK	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700

Foreground colors may be used alone or with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting the foreground mnemonic causes the attribute to default to black.

If `display_attribute` is zero, the occurrence's display attribute is removed, leaving it with the field display attribute. Then, if that occurrence is onscreen, it is displayed with the attribute attached to its field.

This function does not work on an array that is not scrollable. Use `sm_chg_attr` to change the display attribute of an individual field.

RETURNS

–1 if the field isn't found or isn't scrollable, or if `occurrence` is invalid. 0 otherwise.

VARIANTS

```
sm_i_achg(field_name, occurrence, display_attribute);
```

RELATED FUNCTIONS

```
sm_chg_attr(field_number, display_attribute);
```

EXAMPLE

```
/* Highlight the data occurrence under the cursor in a
* scrolling array, so that the highlight will move
* with the occurrence rather than staying on the field. */

#include "smdefs.h"

highlight ()
```

```
{
    int field_number;
    int occurrence
    field_number = sm_getcurno ();
    occurrence = sm_occur_no();
    sm_o_achg (field_number, occurrence, RED | REVERSE);
    return 0;
}
```

allget

load screen from the LDB

SYNOPSIS

```
void sm_allget(respect_flag)
int respect_flag;
```

DESCRIPTION

This function copies data from the local data block to fields on the current screen with matching names.

If `respect_flag` is nonzero, this function does not write to fields that already contain data, or that have their MDT bits set. If the flag is zero, all fields are initialized. When this function is called by the JAM run–time system, or by your screen entry function, it does *not* set MDT bits for the fields it initializes.

This function is called automatically by the JAM screen–display logic, unless LDB processing has been turned off using `sm_dd_able`. Application code should not normally need to call it.

RELATED FUNCTIONS

```
sm_dd_able(flag);
sm_lstore();
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* If you open a window using sm_r_window, you want
 * named fields initialized from the LDB, and LDB
 * processing is off, you will need to call sm_allget
 * explicitly. You could use this, e.g., to make the
 * LDB read-only during a certain transaction. */
 sm_dd_able (0);
...

if (sm_r_window ("popup", 5, 24) == 0)
{
    sm_allget (0);
    while (sm_input (IN_DATA) != EXIT)
    {
        ...
    }
    sm_close_window ();
}
```

amt_format

write data to a field, applying currency editing

SYNOPSIS

```
int sm_amt_format(field_number, buffer)
int field_number;
char *buffer;
```

DESCRIPTION

If the specified field has a currency edit, it is applied to the data in `buffer`. If the resulting string is too long for the field, the excess characters are truncated. Then `sm_putfield` is called to write the edited string to the specified field.

If the field has no currency edit, `sm_putfield` is called with the unedited string.

RETURNS

–1 if the field is not found or the occurrence is out of range;
–2 if the edited string does not fit in the field;
0 otherwise.

VARIANTS

```
sm_e_amt_format(field_name, element, buffer);
sm_i_amt_format(field_name, occurrence, buffer);
sm_n_amt_format(field_name, buffer);
sm_o_amt_format(field_number, occurrence, buffer);
```

RELATED FUNCTIONS

```
sm_dtofield(field_number, value, format);
sm_strip_amt_ptr(field_number, inbuf, );
```

EXAMPLE

```
#include "smdefs.h"

/* Write a list of real numbers, stored as character strings, to the
   screen. The first and last fields in the list are tagged with
   special names.*/

int fld, first, last;
extern char *values[]; /* defined elsewhere */

last = sm_n_fldno ("last");
first = sm_n_fldno ("first");
for (fld = first; fld <= last; ++fld)
{
    sm_amt_format (fld, values[fld - first]);
}
```

ascroll

scroll to a given occurrence

SYNOPSIS

```
int sm_ascroll(field_number, occurrence)
int field_number;
int occurrence;
```

DESCRIPTION

This function scrolls the designated field so that the indicated `occurrence` appears there. Synchronized arrays scroll along with the target array.

The field need not be the first element of a scrolling array. You can use this function, for instance, to place the nineteenth occurrence in the third onscreen element of a five–element scrolling array.

If the requested occurrence cannot be placed in the specified field because it is one of the first or last occurrences in a non–circular array, then `sm_ascroll` scrolls the occurrence onto the screen and returns the occurrence number of the occurrence that is actually in the specified field. Two examples illustrate how this works:

If field number 7 is the third element of a non–circular scrolling array and `occurrence` is 1, a call to `sm_ascroll` places occurrence one in the first element of the array and returns 3, the number of the occurrence actually in field 7.

If field number 7 is the first element of a three element non–circular scrolling array and `occurrence` 20 is the last occurrence in the array, `sm_ascroll` scrolls occurrence 20 onto the screen and returns `18`, the number of the occurence in field 7.

RETURNS

–1 if field or occurrence specification is invalid,
occurrence number of the occurrence in the field if the requested occurrence was brought onscreen but not into the requested field.
0 otherwise.

VARIANTS

```
sm_n_ascroll(field_name, occurrence);
```

RELATED FUNCTIONS

```
sm_rscroll(field_number, req_scroll);
sm_t_scroll(field_number);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Scroll the "records" array (and those synchonized with*/
 * it) to the line indicated in another field on the screen */

#define GOTO_LINE PF4

if (sm_input (IN_DATA) == GOTO_LINE)
{
    sm_n_ascroll ("records", sm_n_intval ("line"));
}
```

backtab

backtab to the start of the last unprotected field

SYNOPSIS

```
void sm_backtab();
```

DESCRIPTION

When the cursor is in a field unprotected from tabbing into, but not in the first enterable position, it is moved to the first enterable position of that field. However, if the cursor is in a field with a previous–field edit and one of the fields specified by the edit is unprotected from tabbing, the cursor is moved to the first enterable position of that field. Otherwise, the cursor is moved to the first enterable position of the tab–unprotected field with the next lowest field number. If the cursor is in the first position of the first unprotected field on the screen, or before the first unprotected field on the screen, it wraps backward into the last unprotected field. When there are no unprotected fields, the cursor doesn't move.

If the destination field is shiftable, it is reset according to its justification. The first enterable position depends on the justification of the field and, in fields with embedded punctuation, on the presence of punctuation.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `sm_input`.

This function is called when the JAM logical key BACK is struck.

RELATED FUNCTIONS

```
sm_home();
sm_last();
sm_nl();
sm_tab();
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Back the cursor up if the user strikes a key
 * indicating s/he has made a particular mistake.
 */
int key;

do {
    key = sm_input (IN_DATA);
    if (key == PF5)
```

```
    {
        sm_quiet_err ("OK, start over");
        sm_backtab ();
    }
} while (key != EXIT && key != XMIT);
```

base_fldno

get the field number of the first element of an array

SYNOPSIS

```
int sm_base_fldno(field_number)
int field_number;
```

DESCRIPTION

A base field number is the field number of the first element of an array. Use `sm_base_fldno` to obtain the base field number of an array.

RETURNS

The field number of the base element of the array containing the specified field, or 0 if the field number is out of range.

bel

beep!

SYNOPSIS

```
void sm_bel();
```

DESCRIPTION

Causes the terminal to beep, ordinarily by transmitting the ASCII BEL code to it. If there is a BELL entry in the video file, sm_bel transmits that instead, usually causing the terminal to flash instead of beeping.

Even if there is no BELL entry, use this function instead of sending a BEL, because certain displays use BEL as a graphics character.

Including a **%B** at the beginning of a message displayed on the status line causes this function to be called.

EXAMPLE

```
#include "smdefs.h"

/* Beep if cost is too high. */
if (sm_n_dblval("cost") > 1000.00 )
    sm_bel();
```

bitop

manipulate validation and data editing bits

SYNOPSIS

```
#include "smbitops.h"

int sm_bitop(field_number, action, bit)
int field_number;
int action;
int bit;
```

DESCRIPTION

You can use this function to inspect and modify validation and data editing bits of screen fields, without reference to internal data structures. The first parameter identifies the field to be operated upon.

`action` may include a test and one manipulation from the following table of mnemonics, which are defined in `smbitops.h`:

Mnemonic	*Meaning*
BIT_CLR	Turn bit off
BIT_SET	Turn bit on
BIT_TOGL	Flip state of bit
BIT_TST	Report state of bit

The third parameter is a bit identifier, drawn from the following table:

Character edits				
N_ALL	N_DIGIT	N_YES_NO	N_ALPHA	N_NUMERIC
N_ALPHNUM	N_FCMASK			

Field edits				
N_RTJUST	N_REQD	N_VALIDED	N_MDT	N_CLRINP
N_MENU	N_UPPER	N_LOWER	N_RETENTRY	N_FILLED
N_NOTAB	N_WRAP	N_ADDLEDS	N_EPROTECT	N_TPROTECT
N_CPROTECT	N_VPROTECT	N_ALLPROTECT	N_SELECTED	

The character edits are not, strictly speaking, bits; you cannot toggle them, but the other functions work as you would expect. N_ALLPROTECT is a special value meaning all four protect bits at once.

N_VALIDED and N_MDT are the only bit operations that can apply to individual offscreen and onscreen occurrences. The protection operations can apply to an array as a whole, including offscreen occurrences (see sm_aprotect). All other bit operations are attached to fixed onscreen positions.

The variant sm_n_bitop can take a group name as an argument. The function then affects the group bits.

This function has two additional variants, sm_a_bitop and sm_t_bitop, which perform the requested bit operation on all elements of an array. Their synopsis appear below. If you include BIT_TST, these variants return 1 only if bit is set for *every* element of the array. The variants sm_i_bitop and sm_o_bitop are restricted to N_VALIDED and N_MDT.

action may include both a test and a manipulation. If the manipulation is successful and the test is true, the function returns 1. If the manipulation is successful and the test is false, the function returns 0. If the manipulation is unsuccessful, the function returns a negative value, regardless of the outcome of the test.

RETURNS

–1 if the field or occurrence was not found;
–2 if the action or bit mnemonic is not one of those listed in the table;
–3 if the request was not valid (e.g. – an attempt to set the right justified edit on a word wrapped field);
1 if a BIT_TST was performed and the result is true;
0 otherwise

VARIANTS

```
sm_a_bitop(array_name, action, bit);
```

```
sm_e_bitop(array_name, element, action, bit);
sm_i_bitop(array_name, occurrence, action, bit);
sm_n_bitop(name, action, bit);
sm_o_bitop(field_number, occurrence, action, bit);
sm_t_bitop(array_number, action, bit);
```

EXAMPLE

```
#include "smbitops.h"

/* Check whether a field is validated. If not, place the
 * cursor there. */

if (! sm_n_bitop ("operation", BIT_TST, N_VALIDED))
{
    sm_n_gofield ("operation");
}

/* Make the array "quantities" required. */

sm_t_bitop (sm_n_fldno ("quantities"), BIT_SET, N_REQD);
```

bkrect

set background color of rectangle

SYNOPSIS

```
int sm_bkrect(start_line, start_column, num_of_lines,
      number_of_columns, background_colors)
int start_line;
int start_column;
int num_of_lines;
int number_of_columns;
int background_colors;
```

DESCRIPTION

This function changes the background color of a rectangular area of the current screen. Any fields or elements that begin within the rectangular area will have their background attributes changed to the specified attribute. This means that if there are any fields or elements that are not entirely contained within the rectangular area, a ragged edge results. Display text that falls with in the rectangular area will have its background attribute set.

The arguments `start_line` and `start_column` use a zero–based start line and column. Therefore they can have any value from 0 through one less than the number of lines (or columns) on the screen.

The background color must be one of the mnemonics defined in `smattrib.h` (B_BLACK, B_BLUE, etc.). You can highlight the background color by oring the background color attribute with B_HILIGHT.

RETURNS

–1 if the starting line or column was invalid.
1 if the starting line and column were valid, but the rectangle had to be truncated to fit.
0 if no error.

EXAMPLE

```
/*  "mondrian" Draw some colored squares on the display*/
int colors[] =
{
    B_RED,
    B_BLUE,
    B_WHITE,
    B_CYAN
```

```
};

int
mondrian()
{
    int i;  for (i=0;i<sizeof(colors)/sizeof(int);i++)
    {
        sm_bkrect( (i/2) * 10, (i & 1) * 40, 10, 40, colors[i]);
    }
    return(0);
}
```

blkdrvr

install block mode driver

[illegible]

SYNOPSIS

```
int sm_blkdrvr();
```

DESCRIPTION

This routine installs the correct driver for the driver name in the video file. It is provided in source code form so that the list of "known" drivers can be extended. Refer to the source code for the list of drivers known (if any) by JAM under your operating system.

This routine is called only if `sm_blkinit` has been called and no block mode driver has been previously installed. In that case the entry BLKDRVR in the video file is checked for the name of a "known" driver.

For an application program to use block mode there must be a call to `sm_blkinit`. In addition either a block mode driver must be installed before that call (thereby forcing block mode) or the video file must specify a driver that has been linked in (thereby leaving the final decision to the terminal operator).

If block mode is not desired, `sm_blkinit` should not be called. In that case, `block.o`, this routine, and the two drivers are not linked into the executable.

To add new "known" drivers, simply add them to the list above.

RETURNS

0 if the driver can be installed.
–1 if not.

RELATED FUNCTIONS

```
sm_blkinit();
sm_blkreset();
```

blkinit

initialize (and turn on) block mode terminal

SYNOPSIS

```
int sm_blkinit();
int return_value;
```

DESCRIPTION

This routine must be called by the application program to initiate block mode terminal action. A block mode terminal driver must have been previously installed by `sm_install`.

This routine checks that a block mode terminal driver is installed. If a driver is found, it is called. The driver should return 0 if all is successful.

Generally the return code can be ignored. If the terminal cannot be put into block mode it will still work (possibly better) in interactive mode.

If the driver signifies that all is OK, the global variable `sm_blkcontrol` is set to point to the local block terminal control handler. All Screen Manager calls for block mode support are made through this control routine.

On the first call to the present routine the driver is called with BLK_INIT to perform any required initialization.

On subsequent calls BLK_BLOCK is called instead of BLK_INIT.

RETURNS

return value from driver if one exists.
–1 otherwise.

RELATED FUNCTIONS

```
sm_blkdrvr();
sm_blkreset();
```

blkreset

reset (and turn off) block mode terminal

SYNOPSIS

```
int sm_blkreset();
int return_value;
```

DESCRIPTION

This routine must be called by the application program to reset block mode terminal action. A block mode terminal driver must have been previously installed by `sm_install`.

This routine checks that a block mode terminal driver is installed. If a driver is found, it is called. The driver should return 0 if all is successful.

Generally the return code can be ignored as the terminal is often already in interactive mode. The exception is on those systems that are normally block mode. Many JAM programs rely on the fact that the terminal can be put into interactive mode.

Note that the driver is called with BLK_CHAR, not with BLK_RESET. The only time the driver is called for a full reset is when JAM is about to go to the operating system – either exiting or performing a "shell escape".

RETURNS

return value from driver if one exists.
–1 otherwise.

RELATED FUNCTIONS

```
sm_blkdrvr();
sm_blkinit();
```

c_keyset

close a keyset

SYNOPSIS

```
#include "smsoftk.h"

int sm_c_keyset(scope)
int scope;
```

DESCRIPTION

This function closes the keyset of the given scope. It frees all memory associated with the keyset and marks that scope as free. If the keyset was currently displayed, the keyset labels are changed to reflect the new keyset.

See the keyset chapter of the *Author's Guide* for a detailed explanation of keyset scopes.

Scope Mnemonic from `smsoftk.h`	*Description*
KS_APPLIC	Application scope.
KS_FORM	Form or window scope.
KS_SYSTEM	`jxform` system key sets.

Use `sm_d_keyset` and `sm_r_keyset` to open keysets.

RETURNS

0 if there is no error
–2 if there is no keyset currently at that scope
–3 if the scope is out of range

RELATED FUNCTIONS

```
sm_r_keyset(name, scope);
sm_d_keyset(address, scope);
```

c_off

turn the cursor off

SYNOPSIS

```
void sm_c_off();
```

DESCRIPTION

This function notifies JAM that the normal cursor setting is off. The normal setting is in effect except:

- When a block cursor is in use, as during menu processing, the cursor is off.
- While Screen Manager functions are writing to the display the cursor is off.
- Within certain error message display functions the cursor is on.

If the display cannot turn its cursor on and off (CON and COF entries are not defined in the video file), this function has no effect.

Use sm_c_on to turn the cursor on.

RELATED FUNCTIONS

```
sm_c_on();
```

EXAMPLE

```
sm_err_reset(
  "Verify that the cursor is turned ON");
sm_c_off();
sm_emsg("Verify that the cursor is turned OFF");
sm_c_on();
sm_emsg("Verify that the cursor is turned ON");
```

c_on

turn the cursor on

SYNOPSIS

```
void sm_c_on();
```

DESCRIPTION

This function notifies JAM that the normal cursor setting is on. The normal setting is in effect except:

- When a block cursor is in use, as during menu processing, the cursor is off.
- While Screen Manager functions are writing to the display the cursor is off.
- Within certain error message display functions the cursor is on.

If the display cannot turn its cursor on and off (CON and COF entries are not defined in the video file), this function has no effect.

Use sm_c_off to turn the cursor off.

RELATED FUNCTIONS

```
sm_c_off();
```

EXAMPLE

```
sm_err_reset(
  "Verify that the cursor is turned ON");
sm_c_off();
sm_emsg("Verify that the cursor is turned OFF");
sm_c_on();
sm_emsg("Verify that the cursor is turned ON");
```

c_vis

turn cursor position display on or off

SYNOPSIS

```
void sm_c_vis(display)
int display;
```

DESCRIPTION

Assigning a non–zero value to `display` displays subsequent status line messages with the cursor's position display. This includes background status messages. Messages that would overlap the cursor position display are truncated.

Setting `display` to zero causes subsequent status line messages to be displayed without the cursor's position display.

This function has no effect if the `CURPOS` entry in the video file is not defined. In that case the cursor position display never appears.

JAM uses an asynchronous function and a status line function to perform the cursor position display. If the application has previously installed either of those, this function overrides it.

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Toggle the cursor position display on or off when
* the PF10 key is struck. The first time the key is
* struck, it will go on. */

static int cpos_on = 0;

switch (sm_input(IN_DATA))
{
...
case PF10:
    sm_c_vis (cpos_on ^= 1);
...
}
```

calc

execute a math edit style expression

SYNOPSIS

```
int sm_calc(field_number, occurrence, expression)
int field_number;
int occurrence;
char *expression;
```

DESCRIPTION

Use `sm_calc` to execute a math edit style expression. With this function you can perform mathematical operations that use the contents of one or more fields and then insert the result into a field.

The third parameter `expression` is a math edit style expression. See the **JAM** *Author's Guide* for a complete description on how to create the expression.

The first two parameters, `field_number` and `occurrence` identify the field and occurrence with which the calculation is associated. Normally you do not need to use them and should set them both to 0.

If you want to use relative references to fields in your expression, use the arguments `field_number` and `occurrence` to specify the field to which they should be relative.

If in the event of a math error you want the cursor to move a specific field, specify that field with `field_number`. In addition, if the desired field is an occurrence within an array, specifying the `occurrence` causes the referenced array to scroll to that occurrence.

RETURNS

–1 if a math error occurred.
0 otherwise.

EXAMPLE

```
/* Compute payment due date. */

sm_calc (0, 0, "paymentduedate = @date(shipdate) + 30");
```

cancel

reset the display and exit

SYNOPSIS

```
void sm_cancel(dummy_arg);
int dummy_arg;
```

DESCRIPTION

This function is installed by `sm_initcrt` to be executed if a keyboard interrupt occurs. It calls `sm_resetcrt` to restore the display to the operating system's default state, and exits to the operating system.

If your operating system supports it, you can also install this function to handle conditions that normally cause a program to abort. If a program aborts without calling `sm_resetcrt`, you may find your terminal in an odd state; `sm_cancel` can prevent that.

The parameter `dummy_arg` is a dummy argument. It should have the value zero. The dummy argument allows `sm_cancel` to be used as a signal handler for the C function `signal`.

EXAMPLE

```
/* the following program segment could be found in
 * some error routines */

#include "smdefs.h"
if (error)
{
    sm_quiet_err(
      "fatal error -- can't continue!\n");
    sm_cancel(0);
}


/* The following code can be used on a UNIX system to
 * install sm_cancel() as a signal handler. */

#include "smdefs.h"
#include <signal.h>

signal (SIGTERM, sm_cancel);
```

chg_attr

change the display attribute of a field

SYNOPSIS

```
int sm_chg_attr(field_number, display_attribute)
int field_number;
int display_attribute;
```

DESCRIPTION

Use this function to change the display attribute of an individual field or an element within an array. To change an occurrence attribute so that the attribute moves with the occurrence use `sm_o_achg`.

If the field is part of a scrolling array, then each occurrence may also have a display attribute that overrides the field display attribute when the occurrence arrives onto the screen.

Possible values for `display_attribute` are defined in `smattrib.h`, as shown in the table below:

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Highlights		Background Highlights	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Colors		Background Colors	
BLACK	0000	B_BLACK	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic causes the attribute to default to black.

NOTE: The variant `sm_o_chg_attr` does not take the usual arguments. The second argument is an element rather than an occurrence.

RETURNS

-1 if the field is not found
0 otherwise.

VARIANTS

```
sm_e_chg_attr(field_name, element, display_attribute);
sm_n_chg_attr(field_name, display_attribute);
sm_o_chg_attr(field_number, element, display_attribute);
```

RELATED FUNCTIONS

```
sm_o_achg(field_number, occurrence, display_attribute);
```

EXAMPLE

```
#include "smdefs.h"
/*Given an array of at least four elements with a base*/
/*field number of three, this function*/
/*changes the display attributes of the first four elements.*/

change_elements()
```

```
{
    int element;
    for (element = 1; element <= 4; element++)
    {
        sm_o_chg_attr (3, element, RED | BLINK);
    }
    return (0);
}
```

ckdigit

validate check digit

SYNOPSIS

```
int sm_ckdigit(field_number, field_data, occurrence, modulus,
      minimum_digits)
int field_number;
char *field_data;
int occurrence;
int modulus;
int minimum_digits;
```

DESCRIPTION

This function is called by field validation. It verifies that `field_data` contains the required minimum number of digits terminated by the proper check digit. If not, it posts an error message before returning. It can also be used to check any character string or field. If `field_data` is null, the string to check is obtained from the `field_number` and `occurrence` and an error message is displayed if the string is bad. If `field_number` is zero, no message is posted, but the function's return code indicates whether the string passed its check.

A fuller description of `sm_ckdigit` is included with the source code, which is distributed with JAM.

Note that this function can be replaced by a user-installed check digit function which field validation will call instead. See the chapter on installing functions.

RETURNS

0 If the field contents are available and valid.
–1 If the field contents do not contain the minimum number of digits or the proper check digit.
–2 If `field_data` is a null pointer and the field or occurrence cannot be found.

cl_all_mdts

clear all MDT bits

SYNOPSIS

```
void sm_cl_all_mdts();
```

DESCRIPTION

Clears the MDT (modified data tag) of every occurrence, both onscreen and off, for every field on the current screen.

JAM sets the MDT bit of an occurrence to indicate that it has been modified, either by keyboard entry or by a call to a function like `sm_putfield`, since the screen was first displayed (i.e., after the screen entry function returns).

RELATED FUNCTIONS

```
sm_tst_all_mdts(occurrence);
```

EXAMPLE

```
#include "smdefs.h"

/* Clear MDT for all fields on the screen. Then write */
 * data to the last field, and check that its MDT is */
 * the first one set. */

int occurrence;
int numflds;

sm_cl_all_mdts();
numflds = sm_inquire (SC_NFLDS);
sm_putfield (numflds, "Hello");
if (sm_tst_all_mdts (&occurrence) != numflds)
    sm_err_reset (
    "Something is rotten in the state of Denmark.");
```

cl_unprot

clear all unprotected fields

SYNOPSIS

```
void sm_cl_unprot();
```

DESCRIPTION

Erases onscreen and offscreen data from all fields that are not protected from clearing (CPROTECT). Date and time fields that take system values are re-initialized. Fields with the null edit are reset to their null indicator values.

This function is normally bound to the CLEAR ALL key.

RELATED FUNCTIONS

```
sm_aprotect(field_number, mask);
```

EXAMPLE

```
/* The following code clears all unprotected fields
 * and puts the cursor into the first one. */

sm_cl_unprot ();
sm_home ();
```

clear_array

clear all data in an array

SYNOPSIS

```
int sm_clear_array(field_number)


int sm_1clear_array(field_number)
int field_number;
```

DESCRIPTION

Both functions clear all data from the array containing the field specified by field_number. The value returned by `sm_num_occurs` is changed to zero. The array is cleared even if it is protected from clearing (CPROTECT).

`sm_clear_array` also clears arrays synchronized with the specified array, except for synchronized arrays that are protected from clearing.

`sm_1clear_array` only clears the specified array.

RETURNS

–1 if the field does not exist;
0 otherwise.

VARIANTS

```
sm_n_clear_array(field_name);
sm_n_1clear_array(field_name);
```

RELATED FUNCTIONS

```
sm_aprotect(field_number, mask);
sm_protect(field_number);
```

EXAMPLE

```
/* Clear the entire array of "names" and arrays
 * synchronized with "names". */
sm_n_clear_array("names");

/* Clear the "totals" column of a screen,
 * without clearing arrays synchronized with "totals". */
sm_n_1clear_array("totals");
```

close_window

close current window

SYNOPSIS

```
int sm_close_window();
```

DESCRIPTION

sm_close_window is used to close a window opened by sm_r_window (or variant), sm_r_at_cur (or variant), or sm_mwindow.

The currently open window is erased, and the screen is restored to the state before the window was opened. All data from the window being closed is lost unless LDB processing is active, in which case named fields are copied to the LDB using sm_lstore. Since windows are stacked, the effect of closing a window is to return to the previous window. The cursor reappears at the position it had before the window was opened.

When using the JAM Executive, use sm_jclose to close a form. sm_jclose calls sm_jform to pop the form stack and open the new top form on the stack. In the case of a window, sm_jclose calls sm_close_window to close the window.

RETURNS

-1 if there is no window open, (i.e. if the currently displayed screen is a form or if no screen is displayed).
0 otherwise.

RELATED FUNCTIONS

```
sm_r_window(screen_name, start_line, start_column);
sm_wselect(window_number);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* In a validation function, if the field contains a */
/* special value, open up a window to prompt for a */
/* second value and save it in another field. */

int validate (field, data, occur, bits)
char *data;
int field, occur, bits;
{
```

```
    char buf[256];

    if (bits & VALIDED)
        return 0;

    if (strcmp(data, "other") == 0)
    {
        sm_r_at_cur ("getsecval");
        if (sm_input (IN_DATA) != EXIT)
            sm_getfield (buf, 1);
        else
            buf[0] = 0;
        sm_close_window ();
        sm_n_putfield ("secval", buf);
    }

    return 0;
}
```

copyarray

copy the contents of one array to another

SYNOPSIS

```
int sm_copyarray(destination_fld, source_fld)
int destination_fld;
int source_fld;
```

DESCRIPTION

This routine copies the contents of the array containing `source_fld` into the array containing `destination_fld`. `source_fld` and `destination_fld` are field numbers. They may be the field number of any of element in the respective array.

The developer is responsible for insuring that the arrays are compatible. Data in source array occurrences that are too long for the destination array are truncated without warning. Data in source array occurrences that are shorter than the destination array's field length are blank filled (with respect for justification).

If the source array has more occurrences than the destination array, the data in the extra occurrences are discarded. If the source array has fewer occurrences than the destination array, trailing occurrences in the destination array are cleared of data (but not de–allocated).

`copyarray` sets the MDT bit and clears the VALIDED bit for each destination array occurrence, indicating that the occurrence has been modified and requires validation.

The variant, `sm_n_copyarray`, searches the LDB for either array if the named field is not found on the screen. However, if the destination LDB item has a scope of 1, meaning that it is a constant, then it is not altered and the function returns –1.

RETURNS

–1 if either field is not found or if the destination array in the LDB has a scope of 1.
0 otherwise.

VARIANTS

```
sm_n_copyarray(destination_name, source_name);
```

RELATED FUNCTIONS

```
sm_clear_array(field_number);
sm_getfield(buffer, field_number);
sm_putfield(field_number, data);
```

d_msg_line

display a message on the status line

SYNOPSIS

```
void sm_d_msg_line(message, display_attribute)
char *message;
int display_attribute;
```

DESCRIPTION

The message in `message` is displayed on the status line, with an initial display attribute of `display_attribute`. If the cursor position display is turned on (see `sm_c_vis`), the end of the status line contains the cursor's current row and column. Messages displayed with `sm_d_msg_line` override both background and field status text.

Messages posted with `sm_d_msg_line` are displayed until the status line is cleared by `sm_d_msg_line`. They persist from screen to screen until cleared. Clearing is accomplished by passing `sm_d_msg_line` an empty string for `message` and a `0` for `display_attribute` (See the example). Once cleared, any currently overidden message resumes. The function `sm_d_msg_line` is itself overridden by `sm_err_reset` and related functions, or by the ready/wait message enabled by `sm_setstatus`.

Possible values for `display_attribute` are defined in `smattrib.h`, as shown in the table below:

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Highlights		Background Highlights	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		
Foreground Colors		Background Colors	
BLACK	0000	B_BLACK	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic causes the attribute to default to black.

Several *percent escapes* provide control over the content and presentation of status messages. The character following the percent sign must be in upper–case. Note that, if a message containing percent escapes is displayed before `sm_initcrt` is called, the percent escapes show up in the message.

If a string of the form %Annnn appears anywhere in the message, the hexadecimal number nnnn is interpreted as a display attribute to be applied to the remainder of the message.

The table gives the numeric values of the logical display attributes you need to construct embedded attributes. If you want a digit to appear *immediately* after the attribute change, pad the attribute to 4 digits with leading zeros. If the following character is not a legal hex digit, then leading zeros are unnecessary.

If a string of the form %Kkeyname appears anywhere in the message, keyname is interpreted as a logical key mnemonic, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the %K is stripped out and the mnemonic remains. Key mnemonics are defined in `smkeys.h`; it is of course the name, not the number, that you want here. The mnemonic must be in upper-case.

If the message begins with a %B, JAM beeps the terminal (using `sm_bel`) before issuing the message.

RELATED FUNCTIONS

```
sm_err_reset(message);
sm_msg(column, disp_length, text);
sm_mwindow(text, line, column);
```

EXAMPLE

```
/* The following prompt uses labels for the EXIT and
 * return keys, and underlines crucial words. */

sm_d_msg_line ("Press %KEXIT to %A0027abort%A7, "
    "or %KNL to %A0027continue%A7.");

/* To clear the status line, use: */

sm_d_msg_line ("", 0);
```

dblval

get the value of a field as a real number

SYNOPSIS

```
double sm_dblval(field_number);
int field_number;
```

DESCRIPTION

This function returns the contents of `field_number` as a real number. It calls `sm_strip_amt_ptr` to remove superfluous amount editing characters before converting the data.

RETURNS

The real value of the field is returned.
If the field is not found, the function returns 0.

VARIANTS

```
sm_e_dblval(field_name, element);
sm_i_dblval(field_name, occurrence);
sm_n_dblval(field_name);
sm_o_dblval(field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_dtofield(field_number, value, format);
sm_strip_amt_ptr(field_number, inbuf, );
```

EXAMPLE

```
#include "smdefs.h"

/* Retrieve the value of a starting parameter. */

double param1;

param1 = sm_n_dblval ("param1");
```

dd_able

turn LDB write–through on or off

SYNOPSIS

```
void sm_dd_able(flag)
int flag;
```

DESCRIPTION

During normal JAM processing, named fields in the screen and local data block are kept in sync. When a screen is displayed (and after the screen entry function completes), values are copied in from the LDB; when control passes from the screen (before the screen entry function is executed), values are copied back to the LDB. Normally, when application code reads or writes a value to or from a named field/LDB entry JAM treats the name as a field name unless no such field exists, in which case JAM treats the name as an LDB entry name. During screen entry and exit processing, this logic is reversed in order to preserve the illusion that screen and LDB entries that share the same name also share the same data.

`sm_dd_able` turns this feature off if `flag` is "0" and on if it is "1". The feature is on by default. When it is off, the LDB is never accessed.

EXAMPLE

```
/* Turn LDB write-through off. */

sm_dd_able (0);
```

deselect

deselect a checklist occurrence

SYNOPSIS

```
int sm_deselect(group_name, group_occurrence)
char *group_name;
int group_occurrence;
```

DESCRIPTION

This function allows you to deselect a specific occurrence within a checklist. The group name and occurrence number is used to reference the desired selection. See the *Author's Guide* for a more detailed discussion of groups.

Use `sm_select` to select a group occurrence and `sm_isselected` to check whether or not a particular group occurrence is currently selected.

NOTE: You can not deselect a radio button occurrence. Using `sm_select` on a radio button occurrence automatically deselects the current selection.

RETURNS

–1 arguments do not reference a checklist occurrence.
0 occurrence not previously selected.
1 occurrence previously selected.

RELATED FUNCTIONS

```
sm_isselected(group_name, group_occurrence);
sm_select(group_name, group_occurrence);
```

dicname

set data dictionary name

SYNOPSIS

```
int sm_dicname(dic_name)
char *dic_name;
```

DESCRIPTION

This function names the application's data dictionary, which is *data.dic* by default. It must be called before JAM initialization, in particular before `sm_ldb_init` is called to initialize the local data block from the data dictionary. The argument `dic_name` is a character string giving the file name; JAM searches for it in all the directories in the `SMPATH` variable.

You can achieve the same effect by defining the `SMDICNAME` variable in your setup file equal to the data dictionary name. See the section on setup files in the *Configuration Guide.*

Use the function `sm_pinquire` to find the name of the data dictionary in use.

RETURNS

–1 if it fails to allocate memory to store the name,
0 otherwise.

RELATED FUNCTIONS

```
sm_pinquire(which);
```

EXAMPLE

```
#include "smdefs.h"

/* Set the name of the application's data
* dictionary to /usr/app/common.dic .*/

sm_dicname ("/usr/app/common.dic");
```

disp_off

get displacement of cursor from start of field

SYNOPSIS

```
int sm_disp_off();
```

DESCRIPTION

Returns the difference between the first column of the current field and the current cursor location. This function ignores offscreen data; use `sm_sh_off` to obtain the total cursor offset of a shiftable field.

RETURNS

The difference between cursor position and start of field, or
–1 if the cursor is not in a field.

RELATED FUNCTIONS

```
sm_getcurno();
sm_sh_off();
```

dlength

get the length of a field's contents

SYNOPSIS

```
int sm_dlength(field_number)
int field_number;
```

DESCRIPTION

Returns the length of data stored in `field_number`. The length does not include leading blanks in right justified fields, or trailing blanks in left–justified fields (which are also ignored by `sm_getfield`). It does include data that have been shifted offscreen.

RETURNS

Length of field contents, or
–1 if the field is not found.

VARIANTS

```
sm_e_dlength(field_name, element);
sm_i_dlength(field_name, occurrence);
sm_n_dlength(field_name);
sm_o_dlength(field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_length(field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Save the contents of the "rank" field in a buffer
 * of the proper size. */

char *save_rank;

if ((save_rank = malloc (sm_n_dlength ("rank") + 1)) == 0)
{
    report_error ("malloc error.");
}
else
{
    sm_n_getfield (save_rank, "rank");
}
```

do_region

rewrite part or all of a screen line

SYNOPSIS

```
void sm_do_region(line, column, length, display_attribute, text)
int line;
int column;
int length;
int display_attribute;
char *text;
```

DESCRIPTION

The screen region defined by `line`, `column`, and `length` is rewritten. `Line` and `column` are counted *from zero*, with (0, 0) the upper left–hand corner of the screen.

If `text` is zero, the screen region is redrawn with whatever `display_attribute` has been assigned. If `text` is shorter than `length`, it is padded out with blanks. In either case, the display attribute of the whole area is changed to `display_attribute`.

Possible values for `display_attribute` are defined in `smattrib.h`, as shown in the table below:

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Highlights		Background Highlights	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Colors		Background Colors	
BLACK	0000	B_BLACK	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic causes the attribute to default to black.

The fifth argument, if passed as zero, must be cast, as in:

```
sm_do_region (line, col, length, attrib, (char *)0);
```

EXAMPLE

```
#include "smdefs.h"
#include "smvideo.h"

/* Place a centered text string in a part of the
 * screen where there is (hopefully) no field.
 * The line number is made zero-relative. */

void centerstring (text, line)
char *text;
int line;
{
    int offset, length = strlen (text);

    offset = (*sm_video[V_COLMS] - length) / 2;
    if (offset < 0)
        return;
    sm_do_region (line - 1, offset, length,
              REVERSE | WHITE, text);
}
```

do_uinstalls

install an application's hook functions

SYNOPSIS

```
void sm_do_uinstalls();
```

DESCRIPTION

Hook functions are installed with the library routine `sm_install`. Most developers find it expedient to add their installation code to the function `sm_do_uinstalls` provided in source form in the file `funclist.c`. In some cases, developers may want to call `sm_install` from other points in their applications.

`sm_do_uinstalls()` is generally called by the main routine. The provided source code calls the library function `sm_install` to install dummy function lists. Developers should replace these dummy calls with their own installation calls.

Note that the installation code for the initialization hook function may not be placed in `sm_do_uinstalls`. The initialization function is called from the library routine `sm_initcrt`. When it is called, JAM has not yet allocated its required memory structures, and the physical display characteristics are still untouched by JAM. In general, it is suggested that hook functions be installed after initialization with `sm_initcrt` (it is required in some cases), but clearly this is an exception. The initialization function must be installed before `sm_initcrt` is called. Since `sm_do_uinstalls` is called after `sm_initcrt`, the installation code for the initialization hook function must be called outside of `sm_do_uinstalls`.

RELATED FUNCTIONS

```
sm_install(usage, what_funcs, howmany);
```

doccur

delete occurrences

SYNOPSIS

```
int sm_o_doccur(field_number, occurrence, count)
int field_number;
int occurrence;
int count;
```

DESCRIPTION

NOTE: This function only exists in the `o_` and `i_` variations. There is NO `sm_doccur` since this function only applies to arrays.

This function deletes the data in `count` occurrences beginning with the specified `occurrence`. If the array is scrollable, then it deallocates `count` occurrences. The data in occurrences following the last deleted occurrence are moved up in the array so that there are no gaps. Fewer than `count` occurrences are deleted if the number of remaining allocated occurrences, starting with the referenced occurrence, is less than `count`.

If `count` is negative, occurrences are inserted instead, subject to limitations explained at `sm_ioccur`. The function `sm_ioccur` is normally used to add blank occurrences.

If `occurrence` is zero, the occurrence used is that of `field_number`. If `occurrence` is nonzero, however, it is taken relative to the first field of the array in which `field_number` occurs.

Any clearing–unprotected synchronized arrays have the same operations performed on them as the referenced array.

This function is normally bound to the DELETE LINE key.

RETURNS

–1 if the field or occurrence number was out of range;
–3 if insufficient memory was available;
otherwise, the number of occurrences actually deleted (zero or more).

VARIANTS

```
sm_i_doccur(field_name, occurrence, count);
```

dtofield

write a real number to a field

SYNOPSIS

```
int sm_dtofield(field_number, value, format)
int field_number;
double value;
char *format;
```

DESCRIPTION

The real number `value` is converted to human–readable form, according to `format`, and moved into `field_number` via a call to `sm_amt_format`. If the `format` string is empty, the number of decimal places is taken from a data type edit, if one exists; failing that, from a currency edit, if one exists; or failing that, defaults to 2.

The number of decimal places may be forced to be an arbitrary number n, via rounding, by using the format string "`%.nf`". The format string "`%t.nf`" may be used to truncate instead of to round.

RETURNS

–1 if the field is not found.
–2 if the destination field has a currency edit but the formatted output is too wide for it.
0 otherwise.

VARIANTS

```
sm_e_dtofield(field_name, element, value, format);
sm_i_dtofield(field_name, occurrence, value, format);
sm_n_dtofield(field_name, value, format);
sm_o_dtofield(field_number, occurrence, value, format);
```

RELATED FUNCTIONS

```
sm_amt_format(field_number, buffer);
sm_dblval(field_number);
```

EXAMPLE

```
/* Place the value of pi on the screen, using the
 * formatting attached to the field. */

sm_n_dtofield ("pi", 3.14159, (char *)0);

/* Do it again, using only three decimal places.

sm_n_dtofield ("pi", 3.14159, "%5.3f");
```

e_

variants that take a field name and element number

SYNOPSIS

```
sm_e_...(field_name, element, ...)
char *field_name;
int element;
```

DESCRIPTION

The `e_` variant functions access one element of an array by field name and element number. For a description of any particular function, look under the related function without `e_` in its name. For example, `sm_e_amt_format` is described under `sm_amt_format`.

Despite the fact that they take a field name as argument, these functions do not search the LDB for names not found in the screen because an element number is ambiguous when referring to the LDB.

edit_ptr

get special edit string

SYNOPSIS

```
char *sm_edit_ptr(field_number, edit_type)
int field_number;
int edit_type;
```

DESCRIPTION

This function searches the special edits area of a field or group for an edit of type `edit_type`. The `edit_type` should be one of the following values, which are defined in `smedits.h`:

Edit type	*Contents of edit string*
NAMED	Field name
CPROG	Name of field validation function
FE_CPROG	Name of field entry function
FX_CPROG	Name of field exit function
HELPSCR	Name of help screen
HARDHLP	Name of automatic help screen
HARDITM	Name of automatic item selection screen
ITEMSCR	Name of item selection screen
SUBMENU	Name of pull–down menu screen
TABLOOK	Name of screen for table–lookup validation
NEXTFLD	Next field (contains both primary and alternate fields)
PREVFLD	Previous field (contains both primary and alternate fields)
TEXT	Status line prompt

Edit type	*Contents of edit string*
MEMO1 ... MEMO9	Nine arbitrary user-supplied text strings
JPLTEXT	Attached JPL code
CALC	Math expression executed at field exit
CKDIGIT	Flag and parameters for check digit
FTYPE	Data type for inclusion in structure
RETCODE	Return value for menu or return entry field
CMASK	Regular expression for field validation
CCMASK	Regular expression for character validation
CKBOX	Offset and attribute of checkbox in a group
ALTSC_CPROG	Name of alternate scrolling function
SDATETIME	Date/time field with user format, initialized with system values.
UDATETIME	Date/time field with user format, initialized by the user.
CURRED	Currency field format, see `smedits.h` for details.
NULLFIELD	Null field representation.
RANGEL	Low bound on range; up to 9 permitted
RANGEH	High bound on range; up to 9 permitted
EDT_BITS	Normally for internal use (see `smedits.h` for more information.)

The string returned by `sm_edit_ptr` contains:

- The total length of the string (including the two overhead bytes and any terminators) in its first byte.
- The edit_type code in its second byte.

- The body of the edit in the subsequent bytes. Refer to the source listing for the file `smedits.h` for specific information on how to interpret each individual edit.

If the field has no edit of type `edit_type`, this function returns a null pointer. If a field has multiple edits of one type, such as RANGEH or RANGEL, then each additional edit is added onto the end of the string following the same pattern as the first one. For example, the first byte would contain the length of the string up to the end of the body of the edit of RANGEH. Adding one to this number would give you the byte that contains the length of the string containing information on RANGEL and so forth.

This function is especially useful for retrieving user-defined information contained in MEMO edits.

In the case of groups, the edits `PREVFLD`, `NEXTFLD`, `CPROG`, `FE_CPROG`, and `FE_CPROG` may be used to obtain group information. For the `CKBOX` edit type, use the `sm_n_edit_ptr` variant with the group name instead of the field name.

Further information on using the `sm_edit_ptr` routine can be found in the file `smedits.h`, normally found in the "include" directory.

RETURNS

A pointer to the first (length) byte of the special edit of the field.
0 if the field or edit is not found.

VARIANTS

```
sm_n_edit_ptr(field_name, edit_type);
```

EXAMPLE

```
 #include "smdefs.h"
 #include "smedits.h"

/* Retrieve the contents of the first MEMO edit */
 * of a field named "cost" */

char *memo1;

if ((memo1 = sm_n_edit_ptr ("cost", MEMO1)) == 0)
{
    sm_emsg("No MEMO1 text!");
}
/* move past the length byte to the body of the edit */
else
{
    memo1 = memo1 +2;
}
```

emsg

display an error message and reset the status line without turning on the cursor

SYNOPSIS

```
void sm_emsg(message)
char *message;
```

DESCRIPTION

This function displays `message` on the status line, if it fits, or in a window if it is too long. If the cursor position display has been turned on (see `sm_c_vis`), the end of the status line contains the cursor's current row and column. If the message text would overlap that area of the status line, it is displayed in a window instead. The message remains visible until the operator presses a key. The function's exact behavior in dismissing the message is subject to the error message options; see `sm_option`.

`sm_emsg` is identical to `sm_err_reset`, except that it does not attempt to turn the cursor on before displaying the message. It is similar to `sm_qui_msg`, which inserts a constant string (normally "ERROR:") before the message.

Several *percent escapes* provide control over the content and presentation of status messages. The character following the percent sign must be in upper–case. Note that, if a message containing percent escapes is displayed before `sm_initcrt` is called, the percent escapes show up in the message.

If a string of the form `%A`nnnn appears anywhere in the message, the hexadecimal number nnnn is interpreted as a display attribute to be applied to the remainder of the message. The table gives the numeric values of the logical display attributes you need to construct embedded attributes. If you want a digit to appear *immediately* after the attribute change, pad the attribute to 4 digits with leading zeros. If the following character is not a legal hex digit, then leading zeros are unnecessary.

If a string of the form `%K`keyname appears anywhere in the message, keyname is interpreted as a logical key mnemonic, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the `%K` is stripped out and the mnemonic remains. Key mnemonics are defined in `smkeys.h`; it is of course the name, not the number, that you want here. The mnemonic must be in upper–case.

If the message begins with a `%B`, JAM beeps the terminal (using `sm_bel`) before issuing the message.

If %N appears anywhere in the message, the latter is presented in a pop–up window rather than on the status line, and all occurrences of %N are replaced by new lines.

If the message begins with %W, it is presented in a pop–up window instead of on the status line. The window appears near the bottom center of the screen, unless it would obscure the current field by so doing; in that case, it appears near the top.

If the message begins with %Mu or %Md, JAM ignores the default error message acknowledgement flag and processes (for %Mu) or discards (for %Md) the next character typed.

Possible hex values for display attribute are defined in `smattrib.h`, as shown in the table below:

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Highlights		Background Highlights	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Colors		Background Colors	
BLACK	0000	B_BLACK	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic causes the attribute to default to black.

RELATED FUNCTIONS

```
sm_err_reset(message);
sm_qui_msg(message);
sm_quiet_err(message);
```

EXAMPLE

```
#include "smdefs.h"

sm_emsg("%MdProcessing complete. Press "
    "%A0017any%A7 key to continue.");
```

err_reset

display an error message and reset the status line

SYNOPSIS

```
void sm_err_reset(message)
char *message;
```

DESCRIPTION

The `message` is displayed on the status line until acknowledged it by pressing a key. If `message` is too long to fit on the status line, it is displayed in a window instead. If the cursor position display has been turned on (see `sm_c_vis`), the end of the status line contains the cursor's current row and column. If the message text would overlap that area of the status line, it is displayed in a window instead. The exact behavior of error message acknowledgement is governed by `sm_option`. The initial message attribute is set by `sm_option`, and defaults to blinking.

This function turns the cursor on before displaying the message, and forces off the global flag `sm_do_not_display`. It is similar to `sm_emsg`, which does not turn on the cursor, and to `sm_quiet_err`, which inserts a constant string (normally "ERROR:") before the message.

Several *percent escapes* provide control over the content and presentation of status messages. See `sm_emsg` for details.

RELATED FUNCTIONS

```
sm_emsg(message);
sm_qui_msg(message);
sm_quiet_err(message);
```

EXAMPLE

```
#include "smdefs.h"

/* Let somebody know that his name isn't in the database. */

int validate (field, name, occur, bits)
char *name;
{
    char buf[128];

    if (getrec (name) == 0)
    {
        sprintf (buf, "%%B%s is not in the\
```

```
        database.", name);
        sm_err_reset (buf);
        return -1;
    }

    return 0;
}
```

fi_open

find a file and open it in binary read only mode

SYNOPSIS

```
FILE *sm_fi_open(file_name)
char *file_name;
```

DESCRIPTION

Use this function to open a file in binary read only mode. The file may be a screen file or any other kind of file.

The file name is first sought in the current directory. If that fails, the path given to `sm_initcrt` is checked. Finally the path defined by `SMPATH` is searched.

If the full path name of a file is longer than 80 characters, then the file is skipped.

RETURNS

0 if the file cannot be found in any path.
Else, the file pointer to the open file stream.

RELATED FUNCTIONS

```
sm_fi_path(file_name);
```

fi_path

return the full path name of a file

SYNOPSIS

```
char *sm_fi_path(file_name)
char *file_name;
```

DESCRIPTION

Use this function to find the full path name of a file. The file may be a screen or any other type of file. A pointer to a static buffer containing the file's full path name is returned.

The file name is first sought in the current directory. If that fails, the path given to `sm_initcrt` is checked. Finally the path defined by `SMPATH` is searched.

If the file is found, the full path name is returned to the caller. Since the static buffer used to hold the full path name is shared by several functions, it should be used or copied quickly.

RETURNS

0 if the file cannot be found in any path.
Else, a pointer to a static buffer containing the path.

RELATED FUNCTIONS

```
sm_fi_open(file_name);
```

finquire

obtain information about a field

SYNOPSIS

```
#include "smglobs.h"

int sm_finquire(field_number, which)
int field_number;
int which;
```

DESCRIPTION

Use this function to obtain various information about a field. The variable `which` is a mnemonic that specifies the particular piece of information desired.

Mnemonics for `which` are defined in the file `smglobs.h`. The following values are available:

Mnemonic	*Meaning*
FD_LINE	Line that field is on (zero based).
FD_COLM	Column of field's first position (zero based).
FD_ATTR	Field attributes (see `smattrib.h`).
FD_LENG	Onscreen field length.
FD_ASIZE	Onscreen array size (1 if scalar).
FD_ELT	Onscreen element number.
FD_SHLENG	Shiftable length.
FD_SHINCR	Shift increment.
FD_SHOFS	Current shift offset (number of positions field has been shifted; 0 if shifted to left edge).
FD_SCINCR	Scrolling increment (for Next/Prev page keys).
FD_SCFLAG	Scrolling array circular? (T/F).

Mnemonic	*Meaning*
FD_SCATTR	Scrolling occurrence display attributes set with sm_i_achg; zero if onscreen element attributes are to be used. For sm_i_finquire and sm_o_finquire variants only.
FD_FELT	First onscreen occurrence of scrolling array (1 if scrolled to top).

RETURNS

The value of which if found.
–1 if the field is not found or which is out of range.
0 otherwise.

VARIANTS

```
sm_e_finquire(field_name, element, which);
sm_i_finquire(field_name, occurrence, which);
sm_n_finquire(field_name, which);
sm_o_finquire(field_number, occurrence, which);
```

RELATED FUNCTIONS

```
sm_gp_inquire(group_name, which);
sm_inquire(which);
sm_iset(which, newval);
sm_pinquire(which);
sm_pset(which, newval);
```

EXAMPLE

```
#include <smglobs.h>

void
toggle_blink(field_number)
int field_number;
{
    int attr;
    attr = sm_finquire(field_number,FD_ATTR);
    attr ^= BLINK;
    sm_chg_attr(field_number,attr);
}
```

fldno

get the field number of an array element or occurrence

SYNOPSIS

```
int sm_n_fldno(field_name)
char *field_name;
```

DESCRIPTION

NOTE: This function only exists in the `e_`, `i_`, `n_`, and `o_` variations. There is NO `sm_fldno` since this function determines the field number given other information.

The `e_` variant returns the field number of an array element specified by `field_name` and `element`. If `element` is zero, then `sm_e_fldno` returns the field number of the named field, or the base element of the named array.

The `i_` and `o_` variants return the number of the field containing the specified occurrence if the occurrence is onscreen, or 0 if the occurrence is offscreen.

The `n_` variant returns the field number of a field specified by name, or the base field number of an array specified by name.

RETURNS

0 if the name is not found, if the element number exceeds 1 and the named field is not an array, or if the occurrence is offscreen.
Otherwise, returns an integer between 1 and the maximum number of fields on the current screen that represents the field number.

VARIANTS

```
sm_e_fldno(field_name, element);
sm_i_fldno(field_name, occurrence);
sm_o_fldno(field_number, occurrence);
```

EXAMPLE

```
#include "smdefs.h"

/* Example #1 */

/* Retrieve the field numbers of the first three */
/* elements of the "horses" array. */

int winnum, placenum, shownum;
```

```
winnum = sm_e_fldno ("horses", 1);
placenum = sm_e_fldno ("horses", 2);
shownum = sm_e_fldno  ("horses", 3);


/* Example #2 */
/* Write a list of real numbers, stored as character strings,
   to the screen. The first and last fields in the list are tagged
   with special names.*/

int fld, first, last;
extern char *values[]; /* defined elsewhere */

last = sm_n_fldno ("last");
first = sm_n_fldno ("first");
for (fld = first; fld <= last; ++fld)
{
    sm_amt_format (fld, values[fld - first]);
}
```

flush

flush delayed writes to the display

SYNOPSIS

```
void sm_flush();
```

DESCRIPTION

This function performs delayed writes and flushes all buffered output to the display. It is called automatically via `sm_input` whenever the keyboard is opened and there are no keystrokes available, *i.e.* typed ahead.

Calling this routine indiscriminately can significantly slow execution. As it is called whenever the keyboard is opened, the display is always guaranteed to be in sync before data entry occurs; however, if you want timed output or other non–interactive display, use of this routine is necessary.

RELATED FUNCTIONS

```
sm_flush();
sm_rescreen();
```

EXAMPLE

```
#include "smdefs.h"

/* Update a system time field once per second,
 * until a key is pressed. */

while (!sm_keyhit (10))
{
    sm_n_putfield ("time_now", "");
    sm_flush ();
}

/* ...process the key */
```

form

display a screen as a form

SYNOPSIS

```
int sm_r_form(screen_name)
char *screen_name;

int sm_d_form(screen_address)
char *screen_address;

int sm_l_form(lib_desc, screen_name)
int lib_desc;
char *screen_name;
```

DESCRIPTION

This set of functions is primarily intended to be used by developers who are writing their own executive. These functions do not update the form stack, so it is generally not a good idea to use them with the JAM Executive. To open a form while under the control of the JAM Executive, use a JAM control string or `sm_jform`.

These functions display the named screen as a base form. Bringing up a screen as a form with `sm_d_form`, `sm_l_form`, `sm_r_form` causes the previously displayed form and windows to be discarded, and their memory freed. The new screen is displayed with its upper left–hand corner at the upper left of the display (position (0, 0)).

If an error occurs a return of –1 or –2 means that the previously displayed form is still displayed and may be used. Other negative return codes indicate that the display is undefined. The caller should display another form before using Screen Manager functions.

When you use `sm_r_form` the named screen is sought first in the memory–resident screen list, and if found there is displayed using `sm_d_form`. It is next sought in all the open screen libraries, and if found is displayed using `sm_l_form`. Next it is sought on disk in the current directory; then under the path supplied to `sm_initcrt`; then in all the paths in the setup variable `SMPATH`. If any path exceeds 80 characters, it is skipped. If the entire search fails, this function displays an error message and returns.

You may save processing time by using `sm_d_form` to display screens that are memory–resident. Use `bin2c` to convert screens from disk files, which you can modify using `jxform`, to program data structures you can compile into your application. A memory–resident screen is never altered at run–time, and may therefore be made shareable on systems that provide for sharing read–only data. `sm_r_form` can

also display memory–resident screens, if they are properly installed using `sm_formlist`. Memory–resident screens are particularly useful in applications that have a limited number of screens, or in environments that have a slow disk (*e.g.* MS–DOS). `screen_address` is the address of the screen in memory.

You may also save processing time by using `sm_l_form` to display screens that are in a library. A library is a single file containing many screens (and/or JPL modules and keysets). You can assemble one from individual screen files using the utility `formlib`. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on the number of paths searched.

The library descriptor, `lib_desc`, is an integer returned by `sm_l_open`, which you must call before trying to read any screens from a library. Note that `sm_r_form` also searches any open libraries.

To display a window use `sm_r_at_cur`, `sm_r_window`, or one of their variants.

RETURNS

0 if no error occurred
–1 if the screen file's format is incorrect; previous form still displayed and available
–2 if the screen cannot be found or the maximum allowable number of files is already open; previous form still displayed and available
–4 if, after the screen has been cleared, the screen cannot be successfully displayed because of a read error;
–5 if, after the screen was cleared, the system ran out of memory;

RELATED FUNCTIONS

```
sm_r_window(screen_name, start_line, start_column);
sm_r_at_cur(screen_name);
```

EXAMPLE

```
#include "smdefs.h"
#include <setjmp.h>

/* If an abort condition exists, read in a special
 * form to handle that condition, discarding all
 * open windows. */

extern jmp_buf re_init;

if (sm_isabort (ABT_OFF) > 0)
{
    sm_r_form ("badstuff");
    if (sm_query_msg ("Do you want to continue?") == 'y')
        longjmp (re_init);
    else sm_cancel ();
}
```

formlist

update list of memory–resident files

SYNOPSIS

```
int sm_formlist(ptr_to_form_list)
struct formlist *ptr_to_form_list;
```

DESCRIPTION

This function adds JPL modules, keysets, and screens to the memory resident form list. Each member of the list is a structure giving the name of the JPL module, screen, or keyset, as a character string, and its address in memory. This function is commonly called from `main`. It can be called any number of times from an application program to augment to the memory resident list.

The library functions `sm_r_form`, `sm_r_window`, `sm_r_at_cur`, and `sm_r_keyset` all take a screen or keyset name as a parameter and search for it in the memory–resident list before attempting to read the screen or keyset from disk. The `jpl` command (see the *JPL Guide*) and the function `sm_jplcall` search the memory resident form list when looking for a JPL procedure to execute.

Since no count is given with the list, care must be taken to end the list with a null entry.

To make a JPL module, keyset, or screen memory resident, you can use the `bin2c` utility to create a static C structure initialized with the binary content of the object. You must then compile and link the structure with the application executable. Alternatively, you can read the object into memory after opening it with the function `sm_fi_open`.

RETURNS

–1 if insufficient memory is available for the new list;
0 otherwise.

RELATED FUNCTIONS

`sm_rmformlist();`

EXAMPLE

```
#include "smdefs.h"

/* Following code adds 2 screens to the memory-resident form list. */

struct form_list new_list[] =
```

```
{
    {"new_form1",  new_form1},
    {"new_form2",  new_form2},
    {0,    0}
};

sm_formlist (new_list);
```

fptr

get the content of a field

SYNOPSIS

```
char *sm_fptr(field_number)
int field_number;
```

DESCRIPTION

This routine returns the contents of the field specified by `field_number`. Leading blanks in right–justified fields and trailing blanks in left–justified fields are stripped.

This function shares with several others a pool of buffers where it stores returned data. The value returned by any of them should therefore be processed quickly or copied. `sm_getfield` is not subject to this restriction.

RETURNS

The field contents, or
0 if the field cannot be found.

VARIANTS

```
sm_e_fptr(field_name, element);
sm_i_fptr(field_name, occurrence);
sm_n_fptr(field_name);
sm_o_fptr(field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_getfield(buffer, field_number);
sm_putfield(field_number, data);
```

EXAMPLE

```
#include "smdefs.h"

/* This function reports the contents of a field. */

void report (fieldname)
char *fieldname;
{
    char buf[256], *stuf;
    if ((stuf = sm_n_fptr (fieldname)) == 0)
        return;

    sprintf (buf, "Field '%s' contains '%s'",
        fieldname, stuf);
    sm_emsg (buf);
}
```

ftog

convert field references to group references

SYNOPSIS

```
char *sm_ftog(field_number, group_occurrence)
int field_number;
int *group_occurrence;
```

DESCRIPTION

This function converts field references to group references. Use `sm_i_gtof` to convert them back.

This function returns the name of the group containing the referenced field and inserts its group occurrence number into the address of `occurrence`.

WARNING: This function returns a pointer to internal data. It remains valid only for the duration of the current screen. Do not change the pointer. While the results are unpredictable, it is safe to say they tend towards the dramatic.

RETURNS

A pointer to the group name if found and indirectly through `group_occurrence` the group occurrence number.
0 otherwise and `group_occurrence` is unchanged.

VARIANTS

```
sm_e_ftog(field_name, element, group_occurrence);
sm_i_ftog(field_name, occurrence, group_occurrence);
sm_n_ftog(field_name, group_occurrence);
sm_o_ftog(field_number, occurrence, group_occurrence);
```

RELATED FUNCTIONS

```
sm_i_gtof(group_name, group_occurrence, occurrence);
```

ftype

get the data type and precision of a field

SYNOPSIS

```
int sm_ftype(field_number, precision_ptr)
int field_number;
int *precision_ptr;
```

DESCRIPTION

This function analyzes the edits of a field or LDB entry, and returns data type information. First the "type" (FTYPE) edit is checked, then the "currency" edit, the "date/time" edit, and finally the "character" edit.

Note that this differs from the functionality of sm_rdstruct, sm_wrtstruct, sm_rrecord, and sm_wrecord. These functions only test the type and character edits. They use the currency edit only to determine the precision of a numeric field that has no type edit.

This function returns an integer containing the data type code, plus any applicable flags. To determine the data type code, **and** (bitwise) the returned value with the mask DT_DTYPE. The data type codes and flags are detailed in the tables below.

Data Type Code	*Meaning*
FT_INT	Type edit is *int.*
FT_UNSIGNED	Type edit is *unsigned int*; or no type edit and character edit is *digit.*
FT_SHORT	Type edit is *short int.*
FT_LONG	Type edit is *long int.*
FT_FLOAT	Type edit is *float.*
FT_DOUBLE	Type edit is *double*; or no type edit and character edit is *numeric.*
FT_ZONED	Type edit is *zoned dec.*
FT_PACKED	Type edit is *packed dec.*
DT_CURRENCY	Currency edit

Data Type Code	*Meaning*
DT_DATETIME	Date/time edit
DT_YESNO	Character edit is *yes/no*.
FT_CHAR	No other major type is valid; could mean that type edit is *character* or character edit is *unfiltered, letters only, alphanumeric,* or *regular expression.*

Flag	*Meaning*
DF_NULL	Null edit
DF_REQUIRED	Data required edit (not applicable to LDB)
DF_WRAP	Word wrap edit
DF_OMIT	Type edit is *omit*

Note that FT_OMIT is not listed as one of the data types. A field that has the type edit *omit* returns the data type determined by any of the other edits, as well as a flag indicating that it has the *omit* type edit.

The function puts the precision of float, double and currency values in the precison_ptr argument. If the precision_ptr is 0, then the precision is not returned.

RETURNS

major data type code plus any applicable flags (see tables above).
0 if field is not found

VARIANTS

```
sm_n_ftype(field_number, precision_ptr);
```

EXAMPLE

```
#include "smedits.h"

if ((sm_n_ftype("value",(int*)0) & DT_DTYPE) == FT_DOUBLE)
{
    /* must be a double */
}
```

fval

force field validation

SYNOPSIS

```
int sm_fval(field_number)
int field_number;
```

DESCRIPTION

This function performs all validations on the indicated field or occurrence, and returns the result. If the field is protected against validation, the checks are not performed and the function returns 0; see `sm_aprotect`. Validations are done in the order listed below. Some are skipped if the field is empty, or if its VALIDED bit is already set (implying that it has already passed validation).

Validation	*Skip if valid*	*Skip if empty*
required	y	n
must fill	y	y
regular expression	y	y
range	y	y
check–digit	y	y
date or time	y	y
table lookup	y	y
currency format	y	n*
math expresssion	n	n
field validation	n	n
JPL function	n	n

* The currency format edit contains a skip–if–empty flag; see the *Author's Guide*.

If you need to force a skip–if–empty validation, make the field required. A field with embedded punctuation must contain at least one non–blank non–punctuation character in order to be considered non–empty; otherwise any non blank character makes the field non–empty.

Math expressions, JPL functions and field validation functions are never skipped, since they can alter fields other than the one being validated.

Field validation is performed automatically within `sm_input` when the cursor exits a field via the `TAB` or `NL` logical keys. All fields on a screen are validated when `XMIT` is pressed (see `sm_s_val`). Application programs need call this function only to force validation of other fields.

RETURNS

-2 if the field or occurrence specification is invalid;
-1 if the field fails any validation;
0 otherwise.

VARIANTS

```
sm_e_fval(array_name, element);
sm_i_fval(field_name, occurrence);
sm_n_fval(field_name);
sm_o_fval(field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_n_gval(group_name);
sm_s_val();
```

EXAMPLE

```
#include "smdefs.h"

/* Make sure that the previous field has been
* validated before checking the current one.
*/

validate (fieldnum, data, occurrence, bits)
int fieldnum, occurrence, bits;
char *data;
{
    if (sm_fval (fieldnum - 1))
    {
        /* Place cursor in the previous field
         * and indicate error */
        sm_gofield (fieldnum - 1);
        return 1;
    }
    ...
}
```

getcurno

get current field number

SYNOPSIS

```
int sm_getcurno();
```

DESCRIPTION

This function returns the number of the field in which the cursor is currently positioned. The field number ranges from 1 to the total number of fields in the screen.

RETURNS

Number of the current field, or
0 if the cursor is not within a field.

RELATED FUNCTIONS

```
sm_occur_no();
```

EXAMPLE

```
#include "smdefs.h"

/* Imagine that the screen contains an 8 by 8 array
 * of fields, like a checkerboard. The following code
 * gets the number of the current field and returns
 * the corresponding row and column. */

void get_location (row, column)
int *row, *column;
{
    int fieldnum;

    if ((fieldnum = sm_getcurno ()) == 0)
        *row = *column = -1;
    else
    {
        *row = (fieldnum - 1) / 8 + 1;
        *column = (fieldnum - 1) % 8 + 1;
    }
}
```

getfield

copy the contents of a field

SYNOPSIS

```
int sm_getfield(buffer, field_number)
char buffer[];
int field_number;
```

DESCRIPTION

This function copies the data found in `field_number` to `buffer`. Leading blanks in right-justified fields and trailing blanks in left-justified fields are not copied. The variants that reference a field by name attempt to get data from the corresponding LDB entry if there is no such field on the screen (except that the order is reversed during screen entry/exit processing).

Responsibility for providing a buffer large enough for the field's contents rests with the calling program. This should be at least one greater than the maximum length of the field, taking shifting into account.

In variants that take `name` as an argument, either the name of a field or a group may be used. In the case of groups, `sm_isselected` is preferred to `sm_getfield` for determining whether or not a group occurrence is selected. If `sm_n_getfield` is called on a radio button, the value in `buffer` is the occurrence number of the selected item. If `sm_i_getfield` is called on a checklist, the value in the first occurrence of the array is the number of the first selected item in the group, the value in the second occurrence is the number of the next selected item in the group and so on. If a checklist has, for example, three items selected, the fourth array occurrence will contain a null string.

Note that the order of arguments to this function is different from that to the related function `sm_putfield`.

RETURNS

The total length of the field's contents, or
–1 if the field cannot be found.

VARIANTS

```
sm_e_getfield(buffer, name, element);
sm_i_getfield(buffer, name, occurrence);
sm_n_getfield(buffer, name);
sm_o_getfield(buffer, field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_fptr(field_number);
sm_isselected(group_name, group_occurrence);
sm_putfield(field_number, data);
```

EXAMPLE

```
#include "smdefs.h"

/* Save the contents of the "rank" field in a buffer
 * of the proper size. */

int size;
char *save_rank;

size = sm_n_dlength ("rank");
if ((save_rank = malloc (size + 1)) == 0)
    report_error ("malloc error.");

sm_n_getfield (save_rank, "rank");
```

getjctrl

get control string associated with a key

SYNOPSIS

```
#include "smkeys.h"

char *sm_getjctrl(key, default)
int key;
int default;
```

DESCRIPTION

Each JAM screen contains a table of control strings associated with function keys. JAM also maintains a default table of keys and control strings, which take effect when the current screen has no control string for a function key you press. This table enables you to define system–wide actions for keys. It is initialized from SMINICTRL setup variables. See the section on setup in the *Configuration Guide* for further information.

This function searches one of the tables for key, a logical key mnemonic found in smkeys.h, and returns a the associated control string. If default is zero, the table for the current screen is searched; otherwise, the system–wide table is searched.

RETURNS

A pointer to the control string
0 if none is found.

RELATED FUNCTIONS

```
sm_putjctrl(key, control_string, default);
```

EXAMPLE

```
#include "smdefs.h"
/* Find control string, if any, that is currently
   associated with a particular key. Look first
   for a screen-specific control string. */
char *
find_control (key)
int key;
{
    char *ptr;
    if ( !(ptr = sm_getjctrl (key, 0) ) )
    {
        ptr = sm_getjctrl (key, 1);
    }
    return ptr;
}
```

getkey

get logical value of the key hit

SYNOPSIS

```
#include "smkeys.h"

int sm_getkey();
```

DESCRIPTION

This function gets and interprets keyboard input and returns the logical value to the calling program. Normal characters are returned unchanged. Logical keys are interpreted according to a key translation file for the particular terminal you are using. See the Keyboard Input section in this guide, the Key Translation section in the *Configuration Guide*, and the `modkey` section in the *Utilities Guide*. `sm_getkey` is normally not needed for application programming, since it is called by `sm_input`.

Logical keys include TRANSMIT, EXIT, HELP, LOCAL PRINT, arrows, data modification keys like INSERT and DELETE CHAR, user function keys PF1 through PF24, shifted function keys SPF1 through SPF24, and others. Defined values for all are in `smkeys.h`. A few logical keys, such as LOCAL PRINT and RESCREEN, are processed locally in `sm_getkey` and not returned to the caller.

There is another function called `sm_ungetkey`, which pushes logical key values back on the input stream for retrieval by `sm_getkey`. Since all JAM input routines call `sm_getkey`, you can use it to generate any input sequence automatically. When you use it, calls to `sm_getkey` do not cause the display to be flushed, as they do when keys are read from the keyboard.

There are a number of user–installed functions that may be called by `sm_getkey`. For further information see the section on installing functions in the *Programmer's Guide*.

Finally, there is a mechanism for detecting an externally established abort condition, essentially a flag, which causes JAM input functions to return to their callers immediately. The present function checks for that condition on each iteration, and returns the ABORT key if it is true. See `sm_isabort`.

Application programmers should be aware that JAM control strings are not executed within this function, but at a higher level within the JAM run–time system (i.e., functions that call `sm_getkey`. If you call this function, do not expect function key control strings to work.

The multiplicity of calls to user functions in `sm_getkey` makes it a little difficult to see how they interact, which take precedence, and so forth. In an effort to clarify the

process, we present an outline of `sm_getkey`. The process of key translation is deliberately omitted, for the sake of clarity; that algorithm is presented separately, in the keyboard translation section of the *Programmer's Guide*.

***Step 1

- If an abort condition exists, return the ABORT key.
- If there is a key pushed back by ungetkey, return that.
- If playback is active and a key is available, take it directly to Step 2; otherwise read and translate input from the keyboard. When the keyboard is read, then the asynchronous function (if one is installed) is called during periods of keyboard inactivity.

*** Step 2

- Pass the key to the keychange function. If that function says to discard the key, go back to Step 1; otherwise if an abort condition exists, return the ABORT key.
- If recording is active, pass the key to the recording function.

*** Step 3

- If the routing table says the key is to be processed locally, do so.
- If the routing table says to return the key, return it; otherwise, go back to Step 1.
- If the key is a soft key, return its logical value.

RETURNS

The standard ASCII value of a displayable key; a value greater than 255 (FF hex) for a key sequence in the key translation file.

RELATED FUNCTIONS

```
sm_keyfilter(flag);
sm_ungetkey(key);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Alternate version of sm_query_msg. This version
 * makes up its mind right away. */

int query (text)
char *text;
```

```
{
    int key;

    sm_d_msg_line (text, REVERSE);
    for (;;)
    {
        switch (key = sm_getkey ())
        {
        case XMIT:
        case 'y':
        case 'Y':
            sm_d_msg_line ("", WHITE");
            return 1;
        default:
            sm_emsg ("%Mu So it's 'no'");
            sm_d_msg_line ("", WHITE);
            return 0;
        }
    }
}
```

gofield

move the cursor into a field

SYNOPSIS

```
int sm_gofield(field_number)
int field_number;
```

DESCRIPTION

Positions the cursor to the first enterable position of `field_number`. If the field is shiftable, it is reset.

In a right–justified field, the cursor is placed in the rightmost position and in a left–justified field, in the leftmost. In either case, if the field has embedded punctuation, the cursor goes to the nearest position not occupied by a punctuation character. Use `sm_off_gofield` to place the cursor in position other than that of the first character of a field.

When called to position the cursor in a scrollable array, `sm_o_gofield` and `sm_i_gofield` return an error if the occurrence number passed exceeds by more than 1 the number of allocated occurrences in the specified array. If the desired occurrence is offscreen, it is scrolled on–screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `sm_input`.

If you use gofield to specify the next field in a field validation function, when a user presses TAB to validate the current field, JAM executes the `gofield` in the validation function, and then TABs to the next field. In order to prevent this extra TAB, the validation function must return non–zero. When non–zero is returned by a validation function, the validation bit is not set. You must use sm_bitop to set the bit in this case.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
sm_e_gofield(field_name, element);
sm_i_gofield(field_name, occurrence);
sm_n_gofield(field_name);
```

```
sm_o_gofield(field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_off_gofield(field_number, offset);
```

EXAMPLE

```
#include "smdefs.h"

/* If the combination of this field and the previous
 * one is invalid, go back to the previous for data
 * entry. */

int validate (field, data, occur, bits)
int field, occur, bits;
char *data;
{
    if (bits & VALIDED)
        return 0;

    if (!lookup (data, sm_fptr (field - 1)))
    {
        sm_novalbit (field - 1);
        sm_gofield (field - 1);
        sm_quiet_err ("Lookup failed -\
        please re-enter both items.");
        return 1;
    }
    return 0;
}
```

gp_inquire

obtain information about a group

SYNOPSIS

```
#include "smglobs.h"

int sm_gp_inquire(group_name, which)
char *group_name;
int which;
```

DESCRIPTION

Use this function to obtain various information about group. The variable `which` is a mnemonic that specifies the particular piece of information desired.

Mnemonics for `which` are defined in the file `smglobs.h`. They are:

Mnemonic	*Meaning*
GP_NOCCS	Number of occurrences in the group (sum of number of occurrences of all fields/arrays in group)
GP_FLAGS	Flags

RETURNS

The value of `which`, if found, or
–1 otherwise.

gtof

convert a group name and index into a field number and occurrence

SYNOPSIS

```
int sm_i_gtof(group_name, group_occurrence, occurrence)
char *group_name;
int group_occurrence;
int *occurrence;
```

DESCRIPTION

NOTE: This function only exists in the `i_` variation. There is no `sm_gtof` since groups cannot be referenced by number.

Use this function to convert a group name and group_occurrence into a field number and occurrence. The variable `group_name` is the name of the group and `group_occurrence` is the specific field within the group.

The function returns the field number of the referenced field and inserts the occurrence number into the memory location addressed by `occurrence`.

Using this function allows you to use other JAM library routines to manipulate group fields by converting group references into field references. For instance, if you wanted to access text from a specific field within a group you would need to use `sm_i_gtof` to get the field and occurrence number before you could use the function `sm_o_getfield` to retrieve the text.

RETURNS

The field number if found.
0 otherwise.

RELATED FUNCTIONS

```
sm_ftog(field_number, group_occurrence);
```

gval

force group validation

SYNOPSIS

```
int sm_n_gval(group_name)
char *group_name;
```

DESCRIPTION

NOTE: This function only exists in the `sm_n_gval` variation. There is no `sm_gval` since groups cannot be referenced by number.

Use this function to force the execution of a group's validation function. Use `sm_s_val` to validate all fields and groups on the screen.

RETURNS

–1 if the group fails any validation.
–2 if the group name is invalid.
0 otherwise.

RELATED FUNCTIONS

```
sm_fval(field_number);
sm_s_val();
```

gwrap

get the contents of a wordwrap array

SYNOPSIS

```
int sm_gwrap(buffer, field_number, buffer_length)
char *buffer;
int field_number;
int buffer_length;
```

DESCRIPTION

This function copies the contents of the array specified by `field_number`, one occurrence at a time, into `buffer`, up to the size specified by `buffer_length`. A space is inserted before every non–empty occurrence, except the first.

The variant `sm_o_gwrap` copies the contents of the array, beginning with the specified occurrence.

RETURNS

The length of transferrable data. If this is greater than `buffer_length`, then the data was truncated.

–1 if the field number is invalid or `buffer_length` is ≤ 0.

VARIANTS

```
sm_o_gwrap(buffer, field_number, occurrence, buffer_length);
```

RELATED FUNCTIONS

```
sm_pwrap(field_number, text);
```

hlp_by_name

display help window

SYNOPSIS

```
int sm_hlp_by_name(help_screen)
char *help_screen;
```

DESCRIPTION

The named screen is displayed and processed as a normal help screen, including input processing for the current field (if any).

Refer to the *Author's Guide* for instructions on how to create various kinds of help screens and for details of the behaviour of help screens.

RETURNS

–1 if screen is not found or other error;
1 if data copied from help screen to underlying field;
0 otherwise.

EXAMPLE

```
#include "smdefs.h"
#include "smedits.h"
/* If user tabs out of empty field, find the field's
 * help screen and execute it. Implemented as a
 * validation function. */

nonempty (field, data, occur, bits)
int field, occur, bits;
char *data;
{
    char *helpscreen;

    if (*data == 0)
    {
        if ((helpscreen = sm_edit_ptr (
            field, HELP)) != 0 ||
            (helpscreen = sm_edit_ptr (
        field, HARDHELP)) != 0)
            sm_hlp_by_name (helpscreen + 2);
    }

    return 0;
}
```

home

home the cursor

SYNOPSIS

```
int sm_home();
```

DESCRIPTION

This function moves the cursor to the first enterable position of the first tab–unprotected field on the screen. If the screen has no tab–unprotected fields, the cursor is moved to the first line and column of the topmost screen. However, if you are using the JAM Executive, the cursor may not be visible if there are no tab–unprotected fields.

The cursor is put into a tab–protected field if it occupies the first line and column of the screen and there are no tab–unprotected fields.

This function doesn't immediately trigger field entry, exit, or validation processing. Processing is based on the cursor position when control returns to `sm_input`.

When the JAM logical key HOME is hit, `sm_home` is called.

RETURNS

The number of the field in which the cursor is left, or
0 if the form has no unprotected fields and the home position is not in a protected field.

RELATED FUNCTIONS

```
sm_backtab();
sm_gofield(field_number);
sm_last();
sm_nl();
sm_tab();
```

EXAMPLE

```
#include "smdefs.h"

/* Suppose that at some point the data entry process
 * has gotten fouled up beyond all repair. The
 * following code fragment could be used to start
 * it over. */

/* ... */
sm_cl_unprot ();
sm_home ();
sm_err_reset ("%MuI'm confused! Let's start over.");
/* ... */
```

i_

variants that take a field name and occurrence number

SYNOPSIS

```
sm_i_...(field_name, occurrence, ...)
char *field_name;
int occurrence;
```

DESCRIPTION

The `i_` variants each refer to data by field name and occurrence number. An occurrence is a slot within an array in which data may be stored. Occurrences may be either on or off–screen. Since JAM treats an individual field as an array with one field, even a single non–scrolling field is considered to have one occurrence. The JAM library contains routines that allow you to manipulate individual occurrences during run–time.

If `occurrence` is zero, the reference is always to the current contents of the named field, or to the current contents of the base field of the named array.

For the description of a particular function, look under the related function without `i_` in its name. For example, `sm_i_amt_format` is described under `sm_amt_format`.

If the named field is not part of the screen currently being displayed, these functions attempt to retrieve or change its value in the local data block.

ininames

record names of initial data files for local data block

SYNOPSIS

```
int sm_ininames(name_list)
char *name_list;
```

DESCRIPTION

Use this routine to set up a list of initialization files for local data block entries. The file names in the single string `name_list` should be separated by commas, semicolons or blanks. There may be up to ten file names. You may achieve the same effect by defining the `SMININAMES` variable in your setup file to the list of names. See setup files in the *Configuration Guide* and the Data Dictionary chapter of the *Author's Guide* for details.

The files contain pairs of names and values, which are used to initialize local data block entries by `sm_ldb_init`. This function is called during JAM initialization, so `sm_ininames` should be called before then. White space in the initialization files is ignored, but we suggest a format like the following:

```
"emperor"       "Julius Caesar"
"lieutenant"    "Mark Antony"
"assassin[1]"   "Brutus"
"assassin[2]"   "Cassius"
```

Entries of all scopes may be freely mixed within all files. We recommend, however, that entries be grouped in files by scope if you are planning to use `sm_lreset`. Use `sm_lreset` to clear all entries of a given scope before reinitializing them from a single file.

RETURNS

–5 if insufficient memory is available to store the names;
0 otherwise.

RELATED FUNCTIONS

```
sm_ldb_init();
sm_lreset(file_name, scope);
```

EXAMPLE

```
/* Set up four initialization files. */

sm_ininames ("scope1.ini, scope2.ini,"
    "scope3.ini, scope4.ini");
```

initcrt

initialize the display and JAM data structures

SYNOPSIS

```
int sm_initcrt(path)
char *path;

void sm_jinitcrt(path)
char *path;

void sm_jxinitcrt(path)
char *path;
```

DESCRIPTION

The function `sm_initcrt` is intended for use only with a user–written executive. It is called automatically by the JAM Executive.

`sm_initcrt` must be called at the beginning of screen handling, that is, before any screens are displayed or the keyboard opened for input to a JAM screen. Functions that set options, such as `sm_option`, and those that install functions or configuration files such as `sm_install` or `sm_vinit`, are the only kind that may be called before `sm_initcrt`.

The argument `path` is a directory to be searched for screen files by `sm_r_window` and variants. First the file is sought in the current directory; if it is not there, it is sought in the path supplied to this function. If it is not there either, the paths specified in the environment variable `SMPATH` (if any) are tried. The `path` argument *must* be supplied. If all forms are in the current directory, or if (as JYACC suggests) all the relevant paths are specified in `SMPATH`, an empty string may be passed. After setting up the search path, `sm_initcrt` performs several initializations:

1. It calls a user–defined initialization routine.
2. It determines the terminal type, if possible by examining the environment (`TERM` or `SMTERM`), otherwise by asking the user.
3. It executes the setup files defined by the environment variables `SMVARS` and `SMSETUP`, and reads in the binary configuration files (message, key, and video) specific to the terminal.
4. It allocates memory for a number of data structures shared among JAM library functions.

5. If supported by the operating system, keyboard interrupts are trapped to a routine that clears the display and exits.
6. It initializes the operating system display and keyboard channels, and clears the display.

The functions `sm_jinitcrt` and `sm_jxinitcrt` are called by `jmain.c` and `jxmain.c` respectively for applications that use the JAM Executive. They, in turn, call `sm_initcrt`.

RETURNS

On an error, `sm_initcrt` prints a descriptive message and terminates;
0 if the function executes successfully.

RELATED FUNCTIONS

```
sm_resetcrt();
sm_jresetcrt();
sm_jxresetcrt();
```

EXAMPLE

```
/* To initialize the Screen Manager without supplying
* a path for screens: */

    sm_initcrt ("");
```

input

open the keyboard for data entry and menu selection

SYNOPSIS

```
int sm_input(initial_mode)
int initial_mode;
```

DESCRIPTION

This routine is only used if you are writing your own executive. Use `sm_input` to open the keyboard for either data entry or menu selection.

You specify which mode you wish to be in with the argument `initial_mode`. Possible choices are defined in `smumisc.h`. They are:

- IN_AUTO JAM checks whether you specified the screen to begin menu mode or data entry mode (See *Author's Guide*).
- IN_DATA Start in data entry mode.
- IN_MENU Start in menu mode.

In most cases you will want to use IN_AUTO mode. Use IN_DATA or IN_MENU if you wish to override the setting that you specified via the Screen Editor.

This routine calls `sm_getkey` to get and interpret keyboard entry. While in data entry mode ASCII data is entered into fields on the screen, subject to any restrictions or edits that were defined for the fields. The routine returns to the calling program when it encounters a logical key, when a "return entry" field is filled or tabbed from, or a key with the return bit set in the routing table.

If the logical value returned by `sm_getkey` is TRANSMIT, EXIT, HELP, or a cursor position key, the processing is determined by a routing table. The routing options are set with `sm_keyoption`. See `sm_keyoption` for more information.

This function replaces version 4.0 `sm_choice`, `sm_menu_proc`, and `sm_openkeybd`. These functions only exist in your version 5.0 library for backward compatibility. We strongly suggest that you do not use them in the future.

RETURNS

The key hit by the end–user that terminated the call to `sm_input`, or the first character of the selected menu item.

inquire

obtain value of a global integer variable

SYNOPSIS

```
#include "smglobs.h"

int sm_inquire(which)
int which;
```

DESCRIPTION

This function is used to obtain the current integer value of a global variable. The desired variable is specified by `which`. If the value of `which` is a true/false (the flag is on or off) value then `sm_inquire` returns `1` for true and `0` for false. If you wish to modify a global integer value use `sm_iset`. The permissible values for `which` are defined in `smglobs.h`. The following values are available:

Mnemonic	*Meaning*
I_NODISP	In non–display mode? (T/F). Initially FALSE, setting TRUE causes no further changes to the actual display, although JAM's internal screen image is kept up to date. Error messages will display. This was release 4's `sm_do_not_display` flag.
I_NOMSG	Error message display disabled? (T/F)
I_INSMODE	In insert mode? (T/F).
I_NOWSEL	LDB merge off for `sm_wselect`? (T/F) Normally false. True can be useful for a quick `sm_wselect/sm_wdeselect` pair.
I_INXFORM	In JAM screen editor? (T/F) Field validation routines are generally still called when in editor; they can check this flag to disable certain features.
I_MXLINES	Number of lines available for use by JAM on the hardware display.
I_MXCOLMS	Number of columns available for use by JAM on the hardware display.
I_NLINES	Maximum number of lines available on the current screen, not including the status line.

Mnemonic	*Meaning*
I_NCOLMS	Maximum number of columns available on the current screen, not including the status line.
I_INHELP	Help level of current screen, or 0 if not in help.
I_BSNESS	Screen manager is in control of display? (T/F). (Replaces the release 4 `inbusiness` function).
I_BLKFLGS	Block mode is turned on? (T/F)
SC_VFLINE	First screen line of viewport (0–based).
SC_VFCOLM	First screen column of viewport (0–based).
SC_VNLINE	Number of lines in viewport.
SC_VNCOLM	Number of columns in viewport.
SC_VOLINE	Line offset of viewport.
SC_VOCOLM	Column offset of viewport.
SC_NLINE	Number of lines in screen.
SC_NCOLM	Number of columns in screen.
SC_CLINE	Current line number in screen (zero–based).
SC_CCOLM	Current column number in screen (zero–based).
SC_NFLDS	Number of fields on screen.
SC_NGRPS	Number of groups on screen.
SC_AFLDNO	Number of the field calling a prototyped field function. Corresponds to the first of the four standard arguments passed to a non-prototyped field function.
SC_AFLDOCC	Occurrence number of the field calling a prototyped field function. Corresponds to the third standard argument passed to a non-prototyped field function. The second standard argument, may be obtained from `sm_getfield` or `sm_o_getfield`.

Mnemonic	*Meaning*
SC_AFLDMDT	Bit mask containing contextual information about the validation state of the field and the circumstances under which a prototyped field function was called. Corresponds to the fourth standard argument passed to a non-prototyped field function.
SC_AGRPMDT	Bit mask containing information about the validation state of the group and the circumstances under which a prototyped group function was called. Corresponds to the second of the two standard arguments passed to a non-prototyped group function. The first standard argument, a pointer to the group name, may be obtained by `sm_getcurno` and `sm_ftog` at group entry and exit. Access to the group name at group validation is not supported.
SC_BKATTR	Background attributes of screen.
SC_BDCHAR	Border character of screen.
SC_BDATTR	Border attributes of screen.

RETURNS

If the argument corresponds to an integer global variable, the current value of that variable is returned.

1 true, flag is set to on.
0 false, flag is set to off.
-1 otherwise.

RELATED FUNCTIONS

```
sm_finquire(field_number, which);
sm_gp_inquire(group_name, which);
sm_iset(which, newval);
sm_pinquire(which);
sm_pset(which, newval);
```

EXAMPLE

```
if (sm_inquire(I_BSNESS))
    sm_err_reset("Problem #2!");
else
    fprintf(stderr,"Problem #2!\n");
```

install

install application functions

SYNOPSIS

```
struct fnc_data *sm_install(usage, what_funcs, howmany)
int usage;
struct fnc_data what_funcs[];
int *howmany;
```

DESCRIPTION

This function installs an application routine to be called from JAM library functions, enabling JAM to pass control to your code in the proper type of function context. The use of this function, along with many examples, is fully documented in the Writing and Installing Hook Functions chapter of the *Programmer's Guide*.

As of release 5, JAM supports function prototyping, which permits JAM library functions to be called directly from within control strings and from within JPL. `sm_install` is used to prototype and install these functions. See the Prototyped Functions section of the Writing and Installing Hook Functions chapter of the *Programmer's Guide*.

RETURNS

For single functions: returns the address of a buffer containing (temporarily) a copy of the old function data structure(s), or zero if no function was previously installed.

For lists of functions: returns a pointer to the updated list and also places the number of entries in the new list in the integer addressed by `howmany`.

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"
/* Install two field functions, defined elsewhere. */

extern int field1(), field2();

static struct fnc_data field[] = {
    { "field1", field1, 0, 0, 0, 0 },
    { "field2", field2, 0, 0, 0, 0 }
};
int count;

count = sizeof(field) / sizeof(struct fnc_data);
sm_install (FIELD_FUNC, field, &count);
```

intval

get the integer value of a field

SYNOPSIS

```
int sm_intval(field_number)
int field_number;
```

DESCRIPTION

This function returns the integer value of the data contained in the field specified by `field_number`. Any punctuation characters in the field, except a leading plus or minus sign, are ignored.

If the field is not found the function returns zero. Since a zero could also mean that the contents of the field is a zero, some other method should be used to check for the existence of a field.

RETURNS

The integer value of the specified field.
0 if the field is not found.

VARIANTS

```
sm_e_intval(field_name, element);
sm_i_intval(field_name, occurrence);
sm_n_intval(field_name);
sm_o_intval(field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_itofield(field_number, value);
```

EXAMPLE

```
/* Retrieve the integer value of the
 * "sequence" field. */

int sequence;

sequence = sm_n_intval ("sequence");
```

ioccur

insert blank occurrences into an array

SYNOPSIS

```
int sm_o_ioccur(field_number, occurrence, count)
int field_number;
int occurrence;
int count;
```

DESCRIPTION

NOTE: This function only exists in the `i_` and `o_` variations. There is no `sm_ioccur`, since this function applies only to arrays.

Inserts `count` blank occurrences before the specified occurrence, moving that occurrence and all following occurrences down. If inserting that many would move an occurrence past the end of its array, fewer are inserted. If the array is scrollable, then this function may allocate up to `count` new occurrences. This function never increases the maximum number of occurrences an array can contain; `sm_sc_max` does that. If `count` is negative, occurrences are deleted instead, subject to limitations described in the page for `sm_doccur`. In addition, this function never inserts more blank occurrences than the number of blank occurrences following the last non–blank occurrence (that is, it won't push data off the end of an array).

If `occurrence` is zero, the occurrence used is that of `field_number`. If `occurrence` is nonzero, however, it is taken relative to the first field of the array in which `field_number` occurs.

Any clearing–unprotected synchronized arrays have the same operations performed on them as the referenced array. Synchronized arrays that are protected from clearing remain constant. Therefore, a protected array may be used to number a list of data stored in a non–protected synchronized array as it grows and shrinks.

This function is normally bound to the INSERT LINE key.

RETURNS

–1 if the field or occurrence number is out of range.
–3 if insufficient memory is available.
otherwise, the number of occurrences actually inserted (zero or more).

VARIANTS

```
sm_i_ioccur(field_name, occurrence, count);
```

EXAMPLE

```
#include "smdefs.h"
/* Insert five blank lines at the beginning of
   an array named "amounts". */

sm_i_ioccur ("amounts", 0, 5);
```

is_no

test field for no

SYNOPSIS

```
int sm_is_no(field_number)
int field_number;
```

DESCRIPTION

The first character of the field contents specified by `field_number` is compared with the first letter of the SM_NO entry in the message file, ignoring case. If they match this function returns a 1 for true. If they do not match for any reason, the function returns a 0 for failure. There is no way to tell if the failure is due to a Y in the field or because of some other problem. If you wish to check for a Y response use `sm_is_yes`.

This function is ordinarily used with one–letter fields possessing the yes/no character edit. In this case, the only characters allowed in the field are y, n, or space (which means n). Unlike other functions, `sm_is_no` does *not ignore leading blanks*.

RETURNS

1 if the field's first character matches the first character of the SM_NO entry in the message file.
0 otherwise.

VARIANTS

```
sm_e_is_no(field_name, element);
sm_i_is_no(field_name, occurrence);
sm_n_is_no(field_name);
sm_o_is_no(field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_is_yes(field_number);
```

is_yes

test field for yes

SYNOPSIS

```
int sm_is_yes(field_number)
int field_number;
```

DESCRIPTION

The first character of the field contents specified by `field_number` is compared with the first letter of the `SM_YES` entry in the message file, ignoring case. If they match this function returns a 1 for true. If they do not match for any reason, the function returns a 0 for failure. There is no way to tell if the failure is due to an `N` in the field or because of some other problem. If you wish to check for an `N` response use `sm_is_no`.

This function is ordinarily used with one-letter fields possessing the yes/no character edit. In this case, the only characters allowed in the field are `y`, `n`, or space (which means `n`). Unlike other functions, `sm_is_yes` does *not ignore leading blanks.*

RETURNS

1 if the field's first character matches the first character of the `SM_YES` entry in the message file.
0 otherwise.

VARIANTS

```
sm_e_is_yes(field_name, element);
sm_i_is_yes(field_name, occurrence);
sm_n_is_yes(field_name);
sm_o_is_yes(field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_is_no(field_number);
```

isabort

test and set the abort control flag

SYNOPSIS

```
int sm_isabort(flag)
int flag;
```

DESCRIPTION

Use `sm_isabort` to set the abort flag to the value of `flag`, and return the old value. `flag` must be one of the following as defined in `smumisc.h`:

Flag	*Meaning*
ABT_ON	set abort flag
ABT_OFF	clear abort flag
ABT_DISABLE	turn abort reporting off
ABT_NOCHANGE	do not alter the flag

Abort reporting is intended to provide a quick way out of processing in the JAM library, which may involve nested calls to `sm_input`. The triggering event is the detection of an abort condition by `sm_getkey`, either an ABORT keystroke or a call to this function with ABT_ON (such as from an asynchronous function).

This function enables application code to verify the existence of an abort condition by testing the flag, as well as to establish one.

RETURNS

The previous value of the abort flag.

EXAMPLE

```
#include "smdefs.h"

/* Establish an abort condition */

sm_isabort (ABT_ON);

/* Verify that an abort condition exists, without
 * altering it. */

if (sm_isabort (ABT_NOCHANGE) == ABT_ON)
    ...
```

iset

change value of global integer variable

SYNOPSIS

```
#include "smglobs.h"

int sm_iset(which, newval)
int which;
int newval;
```

DESCRIPTION

JAM has a number of global parameters and settings. This function is used to modify the current value of global integers. The variable to change is specified by `which`. The new value is specified by `newval`. If you wish to get the value of a global integer use `sm_inquire`.

The permissible values for the argument `which` are defined in the header file `smglobs.h`. The following mnemonics are available:

Mnemonic	*Quantity*	*Meaning*
I_INSMODE	0	Enter overtype mode.
	1	Enter insert mode.
I_NOWSEL	0	LDB merge is on for `sm_wselect`.
	1	LDB merge is off for `sm_wselect`. Normally set to 0. 1 is useful for a quick `sm_wselect/sm_wdeselect` pair, for exanmple to update a realtime clock.
I_NODISP	0	Enable updating of display.
	1	Disable updating of display, except for error messages.
I_NOMSG	0	Display error messages.
	1	Don't display error messages.

If you wish to have a process run in the background, you can set both I_NODISP and I_NOMSG to 1.

RETURNS

If which is one of the permissible values, the former value of the appropriate variable is returned.
1 True, the flag was set to on.
0 False, the flag was set to off.
-1 otherwise.

RELATED FUNCTIONS

```
sm_finquire(field_number, which);
sm_gp_inquire(group_name, which);
sm_inquire(which);
sm_pinquire(which);
sm_pset(which, newval);
```

EXAMPLE

```
void
insert_mode(on_off)
int on_off;
{
    sm_iset(I_INSMODE,on_off);
}
```

isselected

determine whether a radio button or checklist occurrence has been selected

SYNOPSIS

```
int sm_isselected(group_name, group_occurrence)
char *group_name;
int group_occurrence;
```

DESCRIPTION

This function lets you check to see whether or not a specific occurrence within a check list or radio button has been selected. The selection is referenced by the group name and occurrence number. If the occurrence is selected, `sm_isselected` returns a 1. A 0 is returned if the occurrence is not selected. See the *Author's Guide* for a more detailed discussion of groups.

Radio button and checklist occurrences are selected by using `sm_select`. Using `sm_select` on a radio button occurrence causes the current selection to be deselected. Checklist occurrences are deselected with `sm_deselect`.

RETURNS

–1 arguments do not reference a checklist or radio button occurrence.
0 not selected.
1 selected.

RELATED FUNCTIONS

```
sm_deselect(group_name, group_occurrence);
sm_getfield(buffer, field_number);
sm_intval(field_number);
sm_select(group_name, group_occurrence);
```

issv

determine if a screen is in the saved list

SYNOPSIS

```
int sm_issv(screen_name)
char *screen_name;
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. This function searches the save list for a single screen and returns 1 if the screen is found (See `sm_svscreen`).

This function is generally called by applications at screen entry to avoid re–acquiring data (via a database query) for previously saved screens. To accomplish this, first use `sm_svscreen` to add the screen to the save list upon screen exit. Next, use `sm_issv` to check the save list upon screen entry. If the screen is on the save list, you know that it has been previously displayed.

RETURNS

1 if the screen is in the saved list.
0 otherwise.

RELATED FUNCTIONS

```
sm_svscreen(screen_list, count);
```

EXAMPLE

```
/* Perform database query only once */
/* on the screen "results". */

if (!sm_issv("results"))
{
    /* do query . . .*/
    sm_svscreen (screen_list, 1);
}
```

itofield

write an integer value to a field

SYNOPSIS

```
int sm_itofield(field_number, value)
int field_number;
int value;
```

DESCRIPTION

The integer passed to `sm_itofield` is converted to characters and placed in the specified field. A number longer than the field is truncated, on the left or right, according to the field's justification, without warning.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
sm_e_itofield(field_name, element, value);
sm_i_itofield(field_name, occurrence, value);
sm_n_itofield(field_name, value);
sm_o_itofield(field_number, occurrence, value);
```

RELATED FUNCTIONS

```
sm_intval(field_number);
```

EXAMPLE

```
/* Find the length of the data in field number 12 */

sm_n_itofield ("count", sm_dlength (12));
```

jclose

close current window or form under **JAM** Executive control

SYNOPSIS

```
int sm_jclose();
```

DESCRIPTION

The active screen is closed, and the display is restored to the state before the screen was opened. `sm_jclose` should only be used when the JAM Executive is in use.

In the case of closing a form, `sm_jclose` pops the form stack and calls `sm_jform` to display the screen on the top of the form stack.

In the case of closing a window, `sm_jclose` calls `sm_close_window`. The effect of closing a window is to return to the previous window on the window stack. The cursor reappears at the same position it had before the window was opened.

RETURNS

-1 if there is no window open, i.e. if the currently displayed screen is a form (or if there is no screen displayed).
0 otherwise.

RELATED FUNCTIONS

```
sm_close_window();
sm_jform(screen_name);
sm_jwindow(screen_name);
```

EXAMPLE

```
#include "smdefs.h"

/* This is an example of a control function attached to
 * the  XMIT key. It validates login and password
 * information. If the login and password are
 * incorrect, the program proceeds to close three of
 * the four "security" windows used for getting a
 * user's login and password information, and the
 * user may again attempt to enter the information.
 * If the password passes, the welcome screen is
 * displayed, and the user may proceed.
```

```
int complete_login(jptr);
char *jptr;
{
    char pass[10];
    sm_n_getfield(pass, "password");
    /*call routine to validate password*/
    if(!check_password(pass))
    {
        /*close current password window*/
        sm_jclose();
        /*close 3rd underlying login window*/
        sm_jclose();
        /*close 2nd underlying login window*/
        sm_jclose();
        /*in bottom window*/
        sm_emsg("Please reenter login and password");
    }
    else
    {
        sm_d_msg_line("Welcome to Security Systems,\
        Inc.");
        /*open welcome screen*/
        sm_jform("Welcome");
    }
    return (0);
}
```

jform

display a screen as a form under **JAM** control

SYNOPSIS

```
int sm_jform(screen_name)
char *screen_name;
```

DESCRIPTION

This function displays a form under **JAM** control. It must be used with the **JAM** Executive. If you are not using the **JAM** Executive, use `sm_r_form` or one of its variants to display a form. If you wish to display a window under **JAM** control, use `sm_jwindow`.

This function displays the named screen as a form. You may close the form with `sm_jclose`, or leave the task to the **JAM** Executive (e.g., when the user presses the EXIT key). Bringing up a screen as a form causes the previously displayed form and windows to be discarded, and their memory freed. The new form is placed on top of the **JAM** form stack.

The difference between `sm_jform` and `sm_r_form`, other than the function arguments, is that only `sm_jform` manipulates the form stack. Since `sm_jform` calls `sm_r_form`, refer to `sm_r_form` for information on other details, such as how the screen to be displayed is found.

The character string `screen_name` uses the same format as that of a **JAM** control string that displays a form. In addition to the screen's name, you may optionally specify the position of the form on the physical display, the size of the viewport, and which portion of the form should be positioned in the viewport's top–left corner. See the Authoring Reference in the *Author's Guide* for details of viewport positioning. The following are all legal strings:

```
sm_jform("form");
```

Display form's first row and column at the top–left corner of the physical display.

```
sm_jform("(20,10)form");
```

Display form's first row and column at the 20th row and 10th column of the physical display.

```
sm_jform("(20,10,15,8)form");
```

Display the first row and column of the form at the 20th row and 10th column of the physical display in viewport that is 15 rows by 8 columns.

A form may be larger than the viewport. If the viewport does not fit on the screen where indicated, **JAM** attempts to place it entirely on the display at a different location. If you

specify a viewport that is larger than the physical display, the viewport will be the size of the physical display. If you wish to change the viewport size after the window is displayed, use `sm_viewport`.

RETURNS

0 if no error occurred.
–1 if the screen file's format is incorrect.
–2 if the screen cannot be found.
–4 if, after the display has been cleared, the screen cannot be successfully displayed because of a read error.
–5 if, after the display was cleared, the system ran out of memory.

RELATED FUNCTIONS

```
sm_r_form(screen_name);
sm_jwindow(screen_name);
```

EXAMPLE

```
#include "smdefs.h"
/* This could be a control function attached to the
 * XMIT key. Here we have completed entering data
 * on the second of several security screens. If
 * the user entered "bypass" into the login, he
 * bypasses the other security screens, and the
 * "welcome" screen is displayed. If the user
 * login is incorrect, the current window is
 * closed, and the user is back at the initial
 * screen (below). Otherwise, the next security
 * window is displayed. */

int getlogin(jptr)
char *jptr;
{
    char password[10];
    sm_n_getfield(password, "password");
    /* check if "bypass" has been entered into login */
    if (strcmp(password,"bypass"))
        sm_jform("welcome");
        /* check if login is valid */
    else if (check_password(password))
    {
        /*close current (2nd) login window */
        sm_jclose();
        sm_emsg("Please reenter login");
    }
    else
        sm_jwindow("login3");
    return (0);
}
```

jplcall

execute a JPL jpl procedure

SYNOPSIS

```
int sm_jplcall(jplcall_text)
char *jplcall_text;
```

DESCRIPTION

This function executes a JPL procedure precisely as if the following JPL statement were executed from within a JPL procedure:

```
jpl jplcall_text
```

For example, if the value of `jplcall_text` were:

```
verifysal :name 50000
```

then

and

```
jpl verifysal :name 50000
```

would be equivalent. See the *JPL Guide* for further information on the JPL `jpl` command.

RETURNS

–1 if the procedure could not be loaded.
Otherwise, the value returned by the JPL procedure.

jplload

execute the JPL load command

SYNOPSIS

```
int sm_jplload(module_name_list)
char *module_name_list;
```

DESCRIPTION

This function is the C interface to the JPL `load` command. Use this command to load one or more modules into memory.

The character string `module_name_list` may be one or more module names. Separate module names with a space.

Calling `sm_jplload` has precisely the same effect as using the JPL `load` command. See the *JPL Guide* for further information on the JPL `load` command.

Use `sm_jplunload` to remove a module from memory.

RETURNS

–1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
sm_jplpublic(module_name_list);
sm_jplunload(module_name);
```

EXAMPLE

```
void
load_modules()
{
    if (sm_jplload("select.jpl insert.jpl delete.jpl"))
        sm_err_reset("Unable to load modules into memory");
}
```

jplpublic

execute the JPL public command

SYNOPSIS

```
int sm_jplpublic(module_name_list)
char *module_name_list;
```

DESCRIPTION

This function is the C interface to the JPL `public` command. Use this command to load one or more modules into memory.

The character string `module_name` may be one or more module names. Separate module names with a space.

Calling `sm_jplpublic` has precisely the same effect as using the JPL `public` command. See the *JPL Guide* for further information on the JPL `public` command.

Use `sm_jplunload` to remove a module from memory.

RETURNS

–1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
sm_jplload(module_name_list);
sm_jplunload(module_name);
```

jplunload

execute the JPL unload command

SYNOPSIS

```
int sm_jplunload(module_name)
char *module_name;
```

DESCRIPTION

This function is the C interface to the JPL `unload` command. Use this command to remove one or more modules from memory. Modules are read into memory by using either `sm_jplpublic` or `sm_jplload` or via the corresponding JPL commands.

Calling `sm_jplunload` has precisely the same effect as using the JPL `unload` command. See the *JPL Guide* for further information on the JPL `unload` command.

The character string `module_name` may be one or more module names. Separate module names with a space.

RETURNS

–1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
sm_jplload(module_name_list);
sm_jplpublic(module_name_list);
```

EXAMPLE

```
void
unload_modules()
{
    if (sm_jplunload("select.jpl insert.jpl delete.jpl"))
        sm_err_reset("Unable to unload modules from memory");
}
```

jtop

start the **JAM** Executive

SYNOPSIS

```
int sm_jtop(screen_name)
char *screen_name;
```

DESCRIPTION

All applications using the **JAM** Executive must include a call to `sm_jtop`. This function starts the **JAM** Executive. The argument `screen_name` is the name of the first screen that your application displays. It is displayed as a form. Once `sm_jtop` is called the **JAM** Executive is in control until the user exits the application.

The **JAM** Executive makes calls to various **JAM** functions that handle all of the tasks needed to control the flow of an application such as opening the keyboard for input, opening and closing forms and windows, and processing all control strings.

If you do not use `sm_jtop` you will have to write your own procedures to control the flow of your application. See the **JAM** Development Overview for a more detailed discussion of the **JAM** Executive.

RETURNS

0 Always.

jwindow

display a window at a given position under **JAM** control

SYNOPSIS

```
int sm_jwindow(screen_name)
char *screen_name;
```

DESCRIPTION

This function displays a window under **JAM** control. It must be used with the **JAM** Executive. If you are not using the **JAM** Executive, use `sm_r_window` or one of its variants to display a window. To display a form under **JAM** control, use `sm_jform`.

This function displays the named screen as a window, by calling `sm_r_window`. You may close the window with a call to `sm_jclose`, or leave the task to the **JAM** Executive (e.g., when the user presses the EXIT key).

Since `sm_jwindow` calls `sm_r_window`, refer to `sm_r_window` for information on how the screen to be displayed is found.

The character string `screen_name` uses a format similar to that of a **JAM** control string that displays a window. Use a single ampersand to specify a stacked window and a double ampersand to specify a sibling window. If the ampersand is omitted, then the screen is opened as a stacked window. In addition to the screen's name, you may optionally specify the position of the window on the physical display, the size of the viewport, as well as which portion of the window should be positioned in the viewport's top–left corner. The positioning and sizing syntax is identical to that of `sm_jform`. See `sm_jform` for examples of acceptable strings.

RETURNS

0 if no error occurred during display of the screen
–1 if the screen file's format is incorrect
–2 if the form cannot be found
–3 if the system ran out of memory but the previous screen was restored

RELATED FUNCTIONS

```
sm_jclose();
sm_jform(screen_name);
sm_r_window(screen_name, start_line, start_column);
```

EXAMPLE

```
#include "smdefs.h"

/* This could be a control function attached to the
 * XMIT key. Here we have completed entering data
 * on the second of several security screens. If
 * the user entered "bypass" into the login, he
 * bypasses the other security screens, and the
 * "welcome" screen is displayed. If the user
 * login is incorrect, the current window is
 * closed, and the user is back at the initial
 * screen (below). Otherwise, the next security
 * window is displayed. */


int getlogin(jptr)
char *jptr;
{
    char password[10];
    sm_n_getfield(password, "password");
    /* check if "bypass" has been entered into
     * login */
    if (strcmp(password,"bypass"))
        sm_jform("welcome");
        /* check if login is valid */
    else if (check_password(password))
    {
        /*close current (2nd) login window */
        sm_jclose();
        sm_emsg("Please reenter login");
    }
    else
        sm_jwindow("login3");
    return (0);
}
```

key_integer

get the integer value of a logical key mnemonic

SYNOPSIS

```
#include "smkeys.h"

int sm_key_integer (key)
char *key;
```

DESCRIPTION

This function returns the integer value of a JAM logical key mnemonic. The value is obtained from the file `smkeys.h`. This function is useful in cases where a function requires the integer value of a key, but cannot access the include files, as in a prototyped function called from JPL. The following table lists the logical key mnemonics:

Logical Key Mnemonics							
EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL	BACK
HOME	DELE	INS	LP	FERA	CLR	SPGU	SPGD
LSHF	RSHF	LARR	RARR	DARR	UARR	REFR	EMOH
INSL	DELL	ZOOM	SFTS	MTGL	VWPT	MOUS	
PF1-PF24		SPF1-SPF24		APP1-APP24		SFT1-SFT24	

RETURNS

the integer value of the logical key mnemonic.
0 if the mnemonic is not found.

RELATED FUNCTIONS

```
sm_keylabel(key);
```

EXAMPLE

The following example is from JPL. It sets the newline key to act as the tab key. The functions `sm_key_integer` and `sm_keyoption` must be prototyped in order to be called from a JPL procedure.

```
vars ret x y
retvar ret

call sm_key_integer "NL"
cat x ret

call sm_key_integer "TAB"
cat y ret

call sm_keyoption :x 2 :y
return
```

keyfilter

control keystroke record/playback filtering

[illegible]

SYNOPSIS

```
int sm_keyfilter(flag)
int flag;
```

DESCRIPTION

This function turns the keystroke record/playback mechanism of `sm_getkey` on (`flag` = 1) or off (`flag` = 0). If no key recording or playback function has been installed, turning the mechanism on has no effect.

It returns a flag indicating whether recording was previously on or off.

RETURNS

The previous value of the filter flag.

RELATED FUNCTIONS

`sm_getkey();`

EXAMPLE

```
/* Disable key recording and playback. */

sm_keyfilter (0);
```

keyhit

test whether a key has been typed ahead

SYNOPSIS

```
int sm_keyhit(interval)
int interval;
```

DESCRIPTION

This function checks whether a key has already been hit; if so, it returns 1 immediately. If not, it waits for the indicated interval and checks again. The key (if any is struck) is *not* read in, and is available to the usual keyboard input routines.

`interval` is in tenths of seconds; the exact length of the wait depends on the granularity of the system clock, and is hardware– and operating–system dependent. JAM uses this function to decide when to call the user–supplied asynchronous function.

If the operating system does not support reads with timeout, this function ignores the interval and only returns 1 if a key has been typed ahead.

RETURNS

0 if no key is available,
non–0 otherwise.

RELATED FUNCTIONS

`sm_getkey();`

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* The following code adds one asterisk per second to
 * a danger bar, until somebody presses EXIT. */

static char *danger_bar = "************************";
int k;

sm_d_msg_line ("You have 25 seconds to find the\
EXIT key.", WHITE);
/* Clear the danger bar area
sm_do_region (5, 10, 25, WHITE, ""); */

for (k = 1; k <= 25; ++k)
```

```
{
    sm_flush ();
}   if (sm_keyhit (10))
    {
        if (sm_getkey () == EXIT)
            break;
    }
    sm_do_region (5, 10, k, WHITE, danger_bar);

if (k <= 25)
    sm_d_msg_line ("%BCongratulations! you win!");
else
    sm_err_reset ("Sorry, you lose.");
```

keyinit

initialize key translation table

[illegible]

SYNOPSIS

```
int sm_keyinit(key_address)
char *key_address;
```

DESCRIPTION

This routine is called by `sm_initcrt` as part of the initialization process, but it can also be called by an application program (either before or after `sm_initcrt`) to install a memory–resident key translation file.

To install a memory–resident key translation file, `key_address` must contain the address of a key translation table created using the `key2bin` and `bin2c` utilities.

RETURNS

0 if the key file is successfully installed.
Program exit if the key file is invalid.

VARIANTS

```
sm_n_keyinit(key_file);
```

keylabel

get the printable name of a logical key

SYNOPSIS

```
#include "smkeys.h"

char *sm_keylabel(key)
int key;
```

DESCRIPTION

Returns the label defined for `key` in the key translation file; the label is usually what is printed on the key on the physical keyboard. If there is no such label, returns the name of the logical key from the following table. Here is a list of key mnemonics:

Logical Key Mnemonics							
EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL	BACK
HOME	DELE	INS	LP	FERA	CLR	SPGU	SPGD
LSHF	RSHF	LARR	RARR	DARR	UARR	REFR	EMOH
INSL	DELL	ZOOM	SFTS	MTGL	VWPT	MOUS	
PF1-PF24		SPF1-SPF24		APP1-APP24		SFT1-SFT24	

If the key code is invalid (not one defined in `smkeys.h`), this function returns an empty string.

RETURNS

A string naming the key, or an empty string if it has no name.

EXAMPLE

```
#include "smkeys.h"

/* Put the name of the TRANSMIT key into a field
 * for help purposes. */

char buf[80];

sprintf (buf, "Press %s to commit the transaction.",
    sm_keylabel (XMIT));
sm_n_putfield ("help", buf);
```

keyoption

set cursor control key options

SYNOPSIS

```
#include "smkeys.h"

int sm_keyoption(key, mode, newval)
int key;
int mode;
int newval;
```

DESCRIPTION

Use `sm_keyoption` to alter at run-time the behavior of `sm_input` when a particular key is pressed. The default values for key options are built in to JAM. This function only works with cursor control keys. Cursor control keys include all JAM logical keys, *except* for PF, SPF, and APP keys. See "Key File" in the *Configuration Guide*.

There are three different possible values for `mode`: `KEY_ROUTING`, `KEY_GROUP`, and `KEY_XLATE`. The mnemonics that they use are defined in `smkeys.h`. All of these modes draw on the following values for `key`.

Logical Key Mnemonics							
EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL	BACK
HOME	DELE	INS*	LP*	FERA	CLR	SPGU	SPGD
LSHF	RSHF	LARR	RARR	DARR	UARR	REFR*	EMOH
INSL	DELL	ZOOM	SFTS*	MTGL	VWPT	MOUS	

- `KEY_ROUTING`

Allows access to the EXECUTE and RETURN bits of the routing table. This mode is generally used to disable a key or to control explictly what action is taken when a key is hit. The following mnemonics may be assigned to `newval`:

1. `KEY_IGNORE` Disables `key`. JAM does nothing when `key` is struck.

2. EXECUTE The action normally associated with key is executed. May be ored with RETURN.
3. RETURN No action is performed, but the function returns to the caller in your code. Used to gain direct control of key's action. May be ored with EXECUTE.

- KEY_GROUP

Allows access to the group action bits. Use this function to control the action of the cursor when it is within a group. The following values may be assigned to newval:

1. VF_GROUP Obey group semantics. Hitting key causes the cursor to move to the next field within the group in the indicated direction. If this mnemonic is ored with VF_CHANGE the cursor exits the group in the indicated direction.
2. VF_CHANGE This value has no effect, unless it is ored with VF_GROUP. In this case the cursor exits the group in the indicated direction.
3. 0 Assigning zero to newval causes key to treat a field within a group as if it were not part of a group.
4. VF_OFFSCREEN Offscreen data scrolls onscreen from the direction indicated.
5. VF_NOPROT key moves cursor into a field protected from tabbing.

- KEY_XLATE

Allows access to the cursor table. Use this routine to assign key the action performed by newval. key may be any of the cursor control keys listed in the table above. newval may be any key—logical, function, application, ASCII, etc.

*Note that INS, REFR, SFTS, and LP may not be used with KEY_XLATE.

RETURNS

-1 if some parameter is out of range.
the old value otherwise.

EXAMPLE

```
/*newline_is_xmit: Map the new line key (return or enter on most
  keyboards) to XMIT -or- reset it back to NL.
  Invoke from a control string as:
    ^newline_is_xmit X    To make NL act as XMIT
    ^newline_is_xmit N    To make NL act as NL                    */

int
newline_is_xmit(cs_data)
```

```
char *cs_data;
{
    while (*cs_data && *cs_data != ' ')
        cs_data++; while (*cs_data == ' ')
        cs_data++; if (*cs_data == 'X')
    {
        sm_keyoption(NL,KEY_XLATE,XMIT);
    }
    else
    {
        sm_keyoption(NL,KEY_XLATE,NL);
    }
    return(0);
}
```

keyset

open a keyset

SYNOPSIS

```
#include "smsoftk.h"

int sm_r_keyset(name, scope)
char *name;
int scope;


int sm_d_keyset(address, scope)
char *address;
int scope;


int sm_l_keyset(lib_desc, name, scope);
int lib_desc;
char *name;
int scope;
```

DESCRIPTION

Use `sm_d_keyset`, `sm_r_keyset`, and `sm_l_keyset` to display a keyset. The parameter `name` is the name of the keyset. `scope` must be one of the mnemonics listed in `smsoftk.h`. Application programs normally use scope `KS_APPLIC`. Values for `scope` are defined in `smsoftk.h`. For a more detailed explanation of scope see the Keyset Editor chapter of the *Author's Guide*.

If there is currently a keyset of the specified scope the name of that keyset is compared with the name passed. If they are the same the present routine returns immediately. This means that if you want to "refresh" a keyset with a new copy from disk, you must first close the keyset with a call to `sm_c_keyset`.

If the call is not successful then the current keyset remains displayed and an error message is posted to the end–user, except where noted otherwise.

The most commonly used variant is `sm_r_keyset`. You do not need to know where the keyset resides because `sm_r_keyset` searches for you. It looks first in the memory resident form list, next in any open libraries, then on disk in the directory specified by the argument to `sm_initcrt`, and finally in the directories specified by `SMPATH`. Keyset files may be mixed freely with screen files in the screen list and in libraries.

You may save processing time by using `sm_d_keyset` to display a memory–resident keyset. `address` is a pointer to the keyset in memory. Use the utility `bin2c` to create program data structures, from disk–based keysets, that you can compile into your application.

You may also save processing time by using `sm_l_keyset` to display keysets that are in a library. A library is a single file containing many keysets (and/or JPL modules and screens). You can assemble one from individual keyset files using the utility `formlib`. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on the number of paths searched.

The library descriptor, `lib_desc`, is an integer returned by `sm_l_open`, which you must call before trying to read any keysets from a library. Note that `sm_r_keyset` also searches any open libraries.

To close a keyset use `sm_c_keyset`.

RETURNS

0 If no error occurred during display of the keyset.
–1 If the format incorrect (not a keyset).
–2 if the keyset cannot be found. No message is posted to the end–user.
–3 If the terminal doesn't support soft keys (or scope out of range).
–4 If there is a read error.
–5 If there is a malloc failure.

EXAMPLE

```
#include <smsoftk.h>

extern char new_keys[];

void
load_applic_keys()
{
    if (sm_d_keyset(new_keys,KS_APPLIC))
    {
        sm_emsg("Could not load memory resident soft keys.");
    }
}


#include <smsoftk.h>

void
zap_application_soft_keys()
{
    sm_c_keyset(KS_APPLIC);
}
```

kscscope

query current keyset scope

SYNOPSIS

```
#include "smsoftk.h"

int sm_kscscope();
```

DESCRIPTION

This routine returns the scope of the current keyset or `-1` if no keyset is currently active.

This function can be used to determine whether or not the application–level keyset (as opposed to the screen–level or override–level keyset) is currently displayed.

Values for `scope` are defined in `smsoftk.h`.

RETURNS

Current scope, or
–1 if not found.

RELATED FUNCTIONS

```
sm_ksinq(scope, number_keys, number_rows, current_row,
      maximum_len, keyset_name);
sm_skvinq(scope, value, occurrence, attribute, label1, label2);
```

ksinq

inquire about keyset information

SYNOPSIS

```
#include "smsoftk.h"

int sm_ksinq(scope, number_keys, number_rows, current_row,
      maximum_len, keyset_name)
int scope;
int *number_keys;
int *number_rows;
int *current_row;
int maximum_len;
char *keyset_name;
```

DESCRIPTION

Use this routine to obtain the name, number of rows, number of items within a row, and current row of a keyset currently in memory. You supply the keyset's `scope` and five addresses to hold the information returned by `sm_skinq`. `scope` must be one of the mnemonics defined in `smsoftk.h`.

The function places the number of rows in the keyset in `number_row`, the number of soft keys per row in `number_keys`, and the current row number in `current_row`. The name of the keyset is placed in the pre-allocated buffer `keyset_name`. The size of `keyset_name` is specified by `maximum_len`. If the name of the keyset in longer then `keyset_name`, then `sm_ksinq` fills the buffer to the end without adding a null character, otherwise a null character is added to the end of the string. The null pointer may be used for any or all of the parameters about which you do not desire information.

RETURNS

0 if information is returned.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.

RELATED FUNCTIONS

```
sm_kscscope();
sm_skinq(scope, row, softkey, value, display_attribute, label1,
      label2);
sm_skvinq(scope, value, occurrence, attribute, label1, label2);
```

kslabel

set a soft key label and attribute

SYNOPSIS

```
int sm_kslabel(softkey, label1, label2, attribute);
int softkey;
char *label1;
char *label2;
int attribute;
```

DESCRIPTION

This routine is used to set the label on a soft key. Usually `sm_skset` is a more useful, but this function is provided for developers who wish to display soft key labels outside of JAM control. It is also provided so that programs written before JAM provided support for soft keys may take advantage of simulated soft keys. You need not enable soft key support (in `jmain.c`) to use this function.

The first parameter is the soft key number. It must be between one and the number specified in the video file entry KPAR.

The variables `label1` and `label2` are for the first and second lines of the soft key label, respectively. The label text is limited to the length specified in the video file. If you do not wish to change one of the labels, assign it the null pointer.

The `attribute` is only used if the video file specifies that it is available. Typically, this is true only for terminals using simulated labels. `attribute` is specified by using mnemonics listed in `smattrib.h` If you do not wish to change `attribute`, assign it the value: NORMAL_ATTR.

NOTE: This routine actually performs output. Use `sm_skset` if you wish to gain the benefits of delayed–write.

RETURNS

0 if the label was successfully set
–1 the `softkey` parameter is invalid

RELATED FUNCTIONS

```
sm_skset(scope, row, softkey, value, attribute, label1, label2);
```

ksoff

turn off soft key labels

SYNOPSIS

```
void sm_ksoff();
```

DESCRIPTION

When a keyset is opened with any of the library routines, the labels are automatically displayed. If you do not wish to display the labels at any point within your application, use `sm_ksoff` to turn the display off.

If you wish to turn them the label display back on, use `sm_kson`.

RELATED FUNCTIONS

```
sm_kson();
```

kson

turn on soft key labels

SYNOPSIS

```
void sm_kson();
```

DESCRIPTION

Normally, keyset labels are displayed when a keyset is called up. The only way the display can be turned off is with the library routine, `sm_ksoff`. Use this routine to turn the label display back on.

RELATED FUNCTIONS

```
sm_ksoff();
```

l_close

close a library

SYNOPSIS

```
int sm_l_close(lib_desc)
int lib_desc;
```

DESCRIPTION

Closes the library indicated by `lib_desc` and frees all associated memory. The library descriptor is a number returned by a previous call to `sm_l_open`.

RETURNS

–1 if the library file could not be closed.
–2 if the library was not open.
0 if the library was closed successfully.

RELATED FUNCTIONS

```
sm_l_at_cur(lib_desc, screen_name);
sm_l_form(lib_desc, screen_name);
sm_l_open(lib_name);
sm_l_window(lib_desc, screen_name, start_line, start_column);
```

EXAMPLE

```
/* Bring up a window from a library. */

int ld;

if ((ld = sm_l_open ("myforms")) < 0)
    sm_cancel ();
...
sm_l_at_cur (ld, "popup");
...
sm_l_close (ld);
```

l_open

open a library

SYNOPSIS

```
int sm_l_open(lib_name)
char *lib_name;
```

DESCRIPTION

You must use `sm_l_open` to open a library before you use a JPL module, a keyset, or a screen that is stored in the library. Use the utility `formlib` to create a library. (See the **JAM** *Utilites Guide*).

This routine allocates space in which to store information about the library, leaves the library file open, and returns a descriptor identifying the library. The descriptor may subsequently be used by `sm_l_window` and related functions, to display screens stored in the library. The library can also be referenced implicitly by `sm_r_window`, `sm_r_keyset`, and `sm_jplcall`, as well as related functions, which search all open libraries.

The library file is sought in all the directories identified by `SMPATH` and the parameter to `sm_initcrt`. If you define the `SMFLIBS` variable in your setup file as a list of library names `sm_l_open` is automatically called for those libraries. The `sm_r_` routines then search in the specified libraries.

Several libraries may be kept open at once. This may cause problems on systems with severe limits on memory or simultaneously open files.

RETURNS

–1 if the library cannot be opened or read.
–2 if too many libraries are already open.
–3 if the named file is not a library.
–4 if insufficient memory is available.
Otherwise, a non–negative integer that identifies the library file.

RELATED FUNCTIONS

```
sm_jplcall(jplcall_text);
sm_jplload(module_name_list);
sm_jplpublic(module_name_list);
sm_l_at_cur(lib_desc, screen_name);
sm_l_close(lib_desc);
```

```
sm_l_form(lib_desc, screen_name);
sm_l_window(lib_desc, screen_name, start_line, start_column);
sm_r_at_cur(screen_name);
sm_r_form(screen_name);
sm_r_keyset(name, scope);
sm_r_window(screen_name, start_line, start_column);
```

EXAMPLE

```
/* Prompt for the name of a library until a
 * valid one is found. Assume the memory-resident
 * screen contains one field for entering the library
 * name, with suitable instructions. */

int ld;
extern char libquery[];

if (sm_d_form (libquery) < 0)
    sm_cancel ();
sm_d_msg_line ("Please enter the name of\
your library.");

do {
    sm_input (IN_DATA);
} while ((ld = sm_l_open (sm_fptr (1))) < 0);
```

last

position the cursor in the last field

SYNOPSIS

```
void sm_last();
```

DESCRIPTION

Use this function to place the cursor at the first enterable position of the last tab–unprotected field of the current screen. If the last field unprotected from tabbing is right justified, the cursor is placed in the rightmost position of the field. By the same token, if the last unprotected field is left justified, the cursor is placed in the leftmost position of the field.

Unlike sm_home, sm_last does not reposition the cursor if the screen has no unprotected fields.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to sm_input.

This function is called when the JAM logical key EMOH is struck.

RELATED FUNCTIONS

```
sm_backtab();
sm_home();
sm_nl();
sm_tab();
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Assume the last field must be entered for.
 * confirmation. This code puts the cursor there,
 * after the TRANSMIT key is pressed. */

while (sm_input (IN_DATA) != XMIT)
    ;
sm_unprotect (sm_inquire(I_NUMFLDS));
sm_last ();
sm_input (IN_DATA);
if (sm_is_yes (smgetcurno ())
    /* And so forth. . . */
```

lclear

erase LDB entries of one scope

SYNOPSIS

```
int sm_lclear(scope)
int scope;
```

DESCRIPTION

This function erases the values stored in the local data block for all names having a scope of the argument `scope`. Legal values for scope are between `1` and `9`. Constant variables having scope `1` *can* be erased.

Refer to the LDB chapter of the *Programmer's Guide* for a discussion of the scope of LDB entries.

RETURNS

–1 if scope is invalid.
0 otherwise.

RELATED FUNCTIONS

```
sm_lreset(file_name, scope);
```

EXAMPLE

```
/* Clear out LDB entries of scope 6, which has been
 * assigned to customer information. */
#define CUSTOMER_SCOPE 6

sm_lclear (CUSTOMER_SCOPE);
```

ldb_hash

use hash index for the LDB

SYNOPSIS

```
void sm_ldb_hash();
```

DESCRIPTION

This routine specifies that a hash table should be used to search the local data block. You must call `ldb_hash` before **JAM** initialization, in particular, before you call `sm_ldb_init` to initialize the Local Data Block.

Use of a hash table slightly improves the performance of routines which access the LDB, at the expense of the memory required for the table. This performance improvement includes the LDB merge which is performed the first time a screen with named fields is displayed. The degree of improved performance depends upon the distribution of the names in the LDB, and is greater for LDBs with more entries.

RELATED FUNCTIONS

```
sm_ldb_init();
```

EXAMPLE

```
#include "smdefs.h"

/* create a local data block with a hash index */

sm_ldb_hash();
sm_ldb_init();
```

ldb_init

initialize (or reinitialize) the local data block

SYNOPSIS

```
void sm_ldb_init();
```

DESCRIPTION

This function creates an empty index of named data items by reading the data dictionary, then loads values into them from initialization files. Data Dictionary entries with a scope of 0 are not loaded into the LDB. There is no LDB prior to the first execution of this function.

Selected parts of the LDB, namely those assigned a certain scope, can be reinitialized using `sm_lclear` or `sm_lreset`.

This function is called explicitly in `jmain.c` and `jxmain.c`. Other functions that affect its behavior, such as `sm_dicname` and `sm_ininames`, should be called first.

RELATED FUNCTIONS

```
sm_dicname(dic_name);
sm_ininames(name_list);
sm_lreset(file_name, scope);
```

EXAMPLE

```
/* After a catastrophic application failure,
* reboot the index. */
if (bad_data ())
{
    sm_ldb_init ();
    ...
}
```

leave

prepare to leave a **JAM** application temporarily

SYNOPSIS

```
void sm_leave();
```

DESCRIPTION

At times it may be necessary to leave a **JAM** application temporarily. For example you may need to escape to the command interpreter or to execute some graphics functions. In such a case, the terminal and its operating system channel need to be restored to their normal states.

This function should be called before leaving. It clears the physical screen (but not the internal screen image); resets the operating system channel; and resets the terminal (using the RESET sequence found in the video file).

RELATED FUNCTIONS

```
sm_return();
```

EXAMPLE

```
#include "smdefs.h"

/* Escape to the UNIX shell for a directory listing */

sm_leave ();
system ("ls -l");
sm_return ();
sm_c_off ();
sm_d_msg_line ("Hit any key to continue",
                    BLINK | WHITE);
sm_getkey ();
sm_d_msg_line ("", WHITE);
sm_rescreen ();
```

length

get the maximum length of a field

SYNOPSIS

```
int sm_length(field_number)
int field_number;
```

DESCRIPTION

This function returns the maximum length of the field specified by `field_number`. If the field is shiftable, its maximum shifting length is returned. This length is as defined in the JAM Screen Editor, and has no relation to the current contents of the field. Use `sm_dlength` to get the length of the contents.

RETURNS

Length of the field.
0 if the field is not found.

VARIANTS

```
sm_n_length(field_name);
```

RELATED FUNCTIONS

```
sm_dlength(field_number);
```

EXAMPLE

```
/* Compute the number of blanks left in a
 * right-justified field (number 6), and fill them
 * with asterisks. */

int blanks, k;
char buf[256];

blanks = sm_length (6) - sm_dlength (6);
for (k = 0; k < blanks; ++k)
    buf[k] = '*';
sm_getfield (buf + blanks, 6);
sm_putfield (6, buf);
```

lngval

get the long integer value of a field

SYNOPSIS

```
long sm_lngval(field_number)
int field_number;
```

DESCRIPTION

This function returns the contents of `field_number`, converted to a long integer. All non–digit characters are ignored, except for a leading plus or minus sign.

RETURNS

The long value of the field.
0 if the field is not found.

VARIANTS

```
sm_e_lngval(field_name, element);
sm_i_lngval(field_name, occurrence);
sm_n_lngval(field_name);
sm_o_lngval(field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_intval(field_number);
sm_ltofield(field_number, value);
```

EXAMPLE

```
#include "smdefs.h"

/* Retrieve the number of fish in one particular sea
 * (a big number) from the screen. */

#define MEDITERRANEAN 4
long fish;

fish = sm_e_lngval ("seas", MEDITERRANEAN);
```

lreset

reinitialize LDB entries of one scope

SYNOPSIS

```
int sm_lreset(file_name, scope)
char *file_name;
int scope;
```

DESCRIPTION

This function sets local data block entries to values read from `file_name`. The `scope` must be between 1 and 9. References in the file to LDB entries not belonging to `scope` are ignored. All variables belonging to `scope` are cleared before reinitializing. This means that `sm_lreset` erases variables that are not in the file.

The file may be in the current directory, or in any of the directories listed in the SMPATH environment variable. It contains pairs of names with values, each enclosed in quotes. While all whites space outside the quotes is ignored, we recommend for readability that the file have one name–value pair per line. If an entry has multiple occurrences, it may be subscripted in the file. Here are a few sample pairs:

```
"husband"   "Ronald Reagan"
"wife[1]"   "Jane Wyman"
"wife[2]"   "Nancy Davis"
```

If you plan to use this function, we recommend that you group your variables in separate files by scope. You can use `sm_ininames` to list a number of files for initialization.

RETURNS

–1 if file not found or scope out of range.
0 otherwise.

RELATED FUNCTIONS

```
sm_lclear(scope);
```

EXAMPLE

```
/* Reinitialize LDB entries of scope 6, which has been
 * assigned to customer information. */

#define CUSTOMER_SCOPE 6
#define CUSTOMER_INIT  "customers.ini"

sm_lreset (CUSTOMER_INIT, CUSTOMER_SCOPE);
```

lstore

copy everything from screen to LDB

SYNOPSIS

```
int sm_lstore();
```

DESCRIPTION

This function copies data from the screen to local data block entries with matching names.

The JAM Executive automatically calls `sm_lstore` when bringing up a new screen or before closing a window. This function need not be called by application code except under special circumstances.

RETURNS

–3 if sufficient memory is not available.
0 otherwise.

RELATED FUNCTIONS

```
sm_allget(respect_flag);
```

ltofield

place a long integer in a field

SYNOPSIS

```
int sm_ltofield(field_number, value)
int field_number;
long value;
```

DESCRIPTION

The long integer passed to this routine is converted to human–readable form and placed in `field_number`. If the number is longer than the field, it is truncated without warning, on the right or left depending on the field's justification.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
sm_e_ltofield(field_name, element, value);
sm_i_ltofield(field_name, occurrence, value);
sm_n_ltofield(field_name, value);
sm_o_ltofield(field_number, occurrence, value);
```

RELATED FUNCTIONS

```
sm_itofield(field_number, value);
sm_lngval(field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Set the number of fish in the sea to a
 * smallish number. */

#define MEDITERRANEAN 4

sm_i_ltofield ("seas", MEDITERRANEAN, 14L);
```

m_flush

flush the status line

SYNOPSIS

```
void sm_m_flush();
```

DESCRIPTION

This function forces updates to the status line to be written to the display. This is useful if you want to display the status of an operation with `sm_d_msg_line`, without flushing the entire display as `sm_flush` does.

RELATED FUNCTIONS

`sm_flush();`

EXAMPLE

```
#include "smdefs.h"

/* Process a big pile of records, providing
 * status as we go. */
char buf[80];
int k;

k = 0;
do {
    sprintf (buf, "Processing record %d", k + 1);
    sm_d_msg_line (buf, REVERSE | WHITE);
    sm_m_flush ();
} while (process (records[k++]) >= 0);
```

max_occur

get the maximum number of occurrences

SYNOPSIS

```
int sm_max_occur(field_number)
int field_number;
```

DESCRIPTION

This function returns the maximum number of occurrences that the array can hold as defined in the JAM Screen Editor or by `sm_sc_max`. If you wish to find out the highest occurrence number of an array that actually contains data, use `sm_num_occurs`.

RETURNS

0 if the field designation is invalid.
1 for a non–scrollable single field.
The number of elements in a non–scrollable array.
The maximum number of occurrences in a scrollable array.

VARIANTS

```
sm_n_max_occur(field_name);
```

RELATED FUNCTIONS

```
sm_num_occurs(field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Find the number of occurrences in an array of
 * whole numbers, say numbers of children, and
 * allocate some memory to hold them. */

int *children, howmany;

if ((howmany = sm_n_max_occur ("children")) > 0)
    children = (int *)calloc(howmany, sizeof(int));
```

mnutogl

switch between menu mode and data entry mode on a dual–purpose screen

SYNOPSIS

```
int sm_mnutogl(screen_mode)
int screen-mode;
```

DESCRIPTION

JAM supports the use of a single screen as both a menu and a data entry screen, but the screen must be in one or the other "mode" at any given moment. This function can be used to change the mode of the screen and to test which mode the screen is in currently. The `mode` argument may have one of four values as defined in `smumisc.h`:

Value	*Meaning*
`IN_AUTO`	No action (generally used just to test the return value).
`IN_DATA`	Change the screen to data entry mode.
`IN_MENU`	Change the screen to menu mode.
`IN_TOGL`	Toggle the screen from one mode to the other (akin to the `MTGL` logical key).

This function is similar to the built–in control function `jm_mnutogl`.

RETURNS

Mode that the screen was in before the function was called (`IN_DATA` or `IN_MENU`.)
–1 if the mode specification is invalid.

msg

display a message at a given column on the status line

SYNOPSIS

```
void sm_msg(column, disp_length, text)
int column;
int disp_length;
char *text;
```

DESCRIPTION

The message is merged with the current contents of the status line, and displayed beginning at `column`. `disp_length` gives the number of characters to display.

On terminals with onscreen attributes, the column position may need to be adjusted to allow for attributes embedded in the status line. Refer to `sm_d_msg_line` for an explanation of how to embed attributes and function key names in a status line message.

This function is called by the function that updates the cursor position display (see `sm_c_vis`).

RELATED FUNCTIONS

```
sm_d_msg_line(message, display_attribute);
```

EXAMPLE

```
#include "smdefs.h"

/* This code displays a message, then chops out
 * part of it. */

char *text0 = "       ";
char *text1 = "Message is displayed on the status "
    "line at col 1.";

sm_msg(1, strlen(text1), text1);
sm_msg(12, strlen(text0), text0);
```

msg_get

find a message given its number

SYNOPSIS

```
#include "smerror.h"

char *sm_msg_get(number)
int number;
```

DESCRIPTION

The messages used by JAM library routines are stored in binary message files, which are created from text files using the JAM utility, `msg2bin`. Use `sm_msgread` to load message files for use by this function.

This function takes the `number` of the message desired and returns the message, or a less informative string if the message number cannot be matched.

Messages are divided into classes based on their numbers, with up to 4096 messages per class. The message class is the message number divided by 4096, and the message offset within the class is the message number *modulo* 4096. Predefined JAM message numbers and classes are defined in `smerror.h`.

RETURNS

The desired message, if found
otherwise, the message class and number, as a string

RELATED FUNCTIONS

```
sm_msgfind(number);
sm_msgread(code, class, mode, arg);
```

EXAMPLE

```
#include "smdefs.h"
#include "smerror.h"

/* Assume that an anxious programmer has just
 * typed in the question, "Will my boss like
 * my new program?" This code fragment answers
 * the question. */

sm_n_putfield ("answer", rand() & 1 ?
    sm_msg_get (SM_YES) :
    sm_msg_get (SM_NO));
```

msgfind

find a message given its number

SYNOPSIS

```
#include "smerror.h"

char *sm_msgfind(number)
int number;
```

DESCRIPTION

This function takes the `number` of a Screen Manager message, and returns the message string. It is identical to `sm_msg_get`, except that it returns zero if the message number is not found.

Screen Manager message numbers are defined in `smerror.h`.

RETURNS

The message
0 if the message number is out of range

RELATED FUNCTIONS

```
sm_msg_get(number);
sm_msgread(code, class, mode, arg);
```

EXAMPLE

```
#include "smdefs.h"
#include "smerror.h"

/* print out message #4 */

sprintf (buf, "The message reads: %s\n", sm_msgfind
       (SM_BADKEY));
sm_quiet_err (buf);
```

msgread

read message file into memory

SYNOPSIS

```
#include "smerror.h"

int sm_msgread(code, class, mode, arg)
char *code;
int class;
int mode;
char *arg;
```

DESCRIPTION

Reads a single set of messages from a binary message file into memory, after which they can be accessed using `sm_msg_get` and `sm_msgfind`. The `code` argument selects a single message class from a file that may contain several classes:

Code	*Class*	*Message Type*
SM	SM_MSGS	Screen Manager
FM	FM_MSGS	Screen Editor
JM	JM_MSGS	JAM run-time
JX	JX_MSGS	Data Dictionary & Control Strings
UT	UT_MSGS	Utilities
(blank)		Undesignated user

`class` identifies a class of messages. Classes 0–7 are reserved for user messages, and several classes are reserved to JAM; see `smerror.h`. As messages with the prefix `code` are read from the file, they are assigned numbers sequentially beginning at 4096 times `class`.

`mode` is a mnemonic composed from the following list. The first five indicate where to get the message file; at least one of these must be supplied. The latter four modify the basic action.

Mnemonic	*Action*
MSG_DELETE	Delete the message class and recover its memory.
MSG_DEFAULT	Use the default file defined by the setup variable SMMSGS.
MSG_FILENAME	Use the file named by `arg`.
MSG_ENVIRON	Use the file named in an environment variable named by `arg`.
MSG_MEMORY	Use a memory–resident file whose address is given by arg.
MSG_NOREPLACE	Modifier: do not overwrite previously installed messages.
MSG_DSK	Modifier: leave file open, do not read into memory
MSG_INIT	Modifier: do not use screen manager error reporting.
MSG_QUIET	Modifier: do not report errors.

You can or MSG_NOREPLACE with any mode except MSG_DELETE, to prevent overwriting messages read previously. Error messages are displayed on the status line, if the screen has been initialized by `sm_initcrt`; otherwise, they go to the standard error output. You can or MSG_INIT with the mode to force error messages to standard error. Combining the mode with MSG_QUIET suppresses error reporting altogether.

If you or MSG_DSK with the mode, the messages are not read into memory. Instead the file is left open, and `sm_msg_get` and `sm_msgfind` fetch them from disk when requested. If your message file is large, this can save substantial memory; but you should remember to account for operating system file buffers in your calculations.

`arg` contains the environment variable name for MSG_ENVIRON; the file name for MSG_FILENAME; or the address of the memory–resident file for MSG_MEMORY. It may be passed as zero for other modes.

RETURNS

0 if the operation completed successfully.
1 if the message class was already in memory and the mode included MSG_NOREPLACE.
2 if the mode was MSG_DELETE and the message file was not in memory.
–1 if the mode was MSG_ENVIRON and the environment variable was undefined.
–2 if the mode was MSG_ENVIRON or MSG_FILENAME and the message file could not be read from disk; other negative values if the message file was bad or insufficient memory was available.

RELATED FUNCTIONS

```
sm_msg_get(number);
sm_msgfind(number);
```

EXAMPLE

```
#include "smdefs.h"
#include "smerror.h"

/* This code fragment duplicates the Release 3
 * routine sm_msginit(). */

sm_msginit (msg_file)
char * msg_file;
{
    int mode = (msg_file ? MSG_MEMORY : MSG_DEFAULT |
    MSG_NOREPLACE) | MSG_INIT;

    if ( sm_msgread ("SM", SM_MSGS, mode,
        msg_file) < 0 ||
        sm_msgread ("JM", JM_MSGS, mode,
         msg_file) < 0 ||
        sm_msgread ("FM", FM_MSGS, mode,
        msg_file) < 0 ||
        sm_msgread ("JX", JX_MSGS, mode,
        msg_file) < 0)
    {
        sm_resetcrt();
        exit (RET_FATAL);
    }
    sm_msgread ((char *)0, 0, mode & ~MSG_INIT |
                    MSG_QUIET, msg_file);
    return (0);
}
```

mwindow

display a status message in a window

SYNOPSIS

```
int sm_mwindow(text, line, column)
char *text;
int line;
int column;
```

DESCRIPTION

This function displays `text` in a pop–up window, whose upper left–hand corner appears at `line` and `column`. The line and column are counted from `0`. If `line` is `1`, the top of the window appears on the second line of the display. The window itself is constructed on the fly by the run–time system. No data entry is possible in it, nor is data entry possible in underlying screens as long as it is displayed.

Due to the delayed write feature in **JAM**, you should call `sm_flush` to cause the screen to be updated and the message to be displayed, unless you call `sm_input` directly after the call to `sm_mwindow`. `sm_close_window` may be used to close a window called with `sm_mwindow`.

All the percent escapes for status messages, except **%M** and **%W**, are effective. Refer to `sm_emsg` for a list and full description. If either `line` or `column` is negative, the window is displayed according to the rules given for `sm_r_at_cur`.

RETURNS

–1 if there was a malloc failure.
1 if the text had to be truncated to fit in a window.
0 otherwise.

RELATED FUNCTIONS

```
sm_d_msg_line(message, display_attribute);
```

EXAMPLE

```
/* By judicious use of %N's, it is possible to get
 * your messages centered on the screen when you
 * call sm_mwindow().
 */

void poem ()
```

```
{
    sm_mwindow ("The world is too much with us.\
    Late and soon,%N\
Getting and spending, we lay waste our powers.%N\
Little we see in Nature that is ours;%N\
We have given our hearts away, a sordid boon!%N%N\
The sea that bares her bosom to the Moon,%N\
The winds that will be raging at all hours,%N\
And are up-gathered now like sleeping flowers,%N\
For this, for everything, we are out of tune;%N\
It moves us not. Great God! I'd rather be%N\
A pagan, suckled in a creed outworn;%N\
So might I, standing on this pleasant lea,%N\
Have glimpses that would make me less forlorn:%N\
Catch sight of Proteus rising from the sea,%N\
Or hear old Triton blow his wreathed horn.",
    6, 16);
}
```

n_

variants that take a field name only

SYNOPSIS

```
sm_n_...(field_name, ...)
char *field_name;
```

DESCRIPTION

The n_ functions access a field by means of the field/group name. For a complete description of individual functions, look under the related function without n_ in its name. For example, sm_n_amt_format is described under sm_amt_format. If the named field/group is not on the screen, these functions attempt to access a similarly named entry in the local data block.

name

obtain field name given field number

SYNOPSIS

```
char *sm_name(field_number)
int field_number;
```

DESCRIPTION

Given a field number, `sm_name` returns a pointer to a buffer that contains the field name referenced by `field_number`. This routine shares with several others a pool of buffers where it stores returned data. The value returned by any of these routines should therefore be processed promptly or copied.

RETURNS

Pointer to the name of the field referenced, if found.
Else a pointer to a null string.

next_sync

find next synchronized array

SYNOPSIS

```
int sm_next_sync(field_number)
int field_number;
```

DESCRIPTION

Given a field number, this function finds the next array synchronized with the given field, and returns the field number of the corresponding element in that array. The next synchronized array is defined as the one to the right. If `field_number` is in the rightmost synchronized array, the function returns the corresponding element in the leftmost synchronized array (ie– it wraps around the screen).

RETURNS

The field number of the next synchronized array if there is one.
Otherwise, the field number the function was passed.

nl

position cursor to the first unprotected field beyond the current line

SYNOPSIS

```
void sm_nl();
```

DESCRIPTION

This function moves the cursor to the next occurrence of an array, scrolling if necessary. Unlike the down–arrow, it allocates an empty scrolling occurrence if there are no more allocated occurrences below but the maximum has not yet been exceeded.

If the current field is not scrolling, the cursor is positioned to the first unprotected field, if any, following the current *line* of the form. If there are no unprotected fields beyond the current field, the cursor is positioned to the first unprotected field of the screen.

If the screen has no unprotected fields at all, the cursor is positioned to the first column of the line following the current line. If the cursor is on the last line of the form, it goes to the top left–hand corner of the screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `sm_input`.

This function is ordinarily bound to the RETURN key.

RELATED FUNCTIONS

```
sm_backtab();
sm_home();
sm_last();
sm_tab();
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"
/* Scuttle down a scrolling array until we come
 * to a nonblank item, or run out of array. */
char buf[256];

while (sm_t_scroll (sm_fldnumber + 1) &&
    sm_getfield (buf, sm_fldnumber + 1) == 0)
{
    sm_nl ();
}
```

novalbit

forcibly invalidate a field

SYNOPSIS

```
int sm_novalbit(field_number)
int field_number;
```

DESCRIPTION

Resets the VALIDED bit of the specified field, so that the field is again subject to validation when it is next exited, or when the screen is validated as a whole.

JAM sets a field's VALIDED bit automatically when the field passes all its validations. The bit is initially clear, and is cleared whenever the field is altered by keyboard input or by a library function such as `sm_putfield`.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
sm_e_novalbit(field_name, element);
sm_i_novalbit(field_name, occurrence);
sm_n_novalbit(field_name);
sm_o_novalbit(field_number, occurrence);
```

RELATED FUNCTIONS

```
sm_fval(field_number);
sm_s_val();
```

EXAMPLE

```
#include "smdefs.h"

/* Here is a validation function for a "last_name"
 * field. When it is changed, it marks the
 * "first_name" field, which depends on it,
 * invalid. */

int validate (field, data, occur, bits)
char *data;
{
    if (bits & VALIDED) /* Not really changed */
        return 0;

    sm_n_novalbit ("first_name");
    return 0;
}
```

null

test if field is null

SYNOPSIS

```
int sm_null(field_number)
int field_number;
```

DESCRIPTION

Use `sm_null` to test a field to see whether it has both the null edit and contains the null character string that has been assigned to that field. See null edits in the *Author's Guide*.

RETURNS

1 If the field has the null edit and contains the appropriate null character string.
–1 if the field does not exist.
0 otherwise.

VARIANTS

```
sm_e_null(field_name, element);
sm_i_null(field_name, occurrence);
sm_n_null(field_name);
sm_o_null(field_number, occurrence);
```

num_occurs

find the highest numbered occurrence containing data

SYNOPSIS

```
int sm_num_occurs(field_number)
int field_number;
```

DESCRIPTION

This function returns the highest occurrence number of the array specified by `field_number` that actually contains data. The field number may be that of any field with the array.

Most of the time the highest numbered occurrence containing data is the same as the number of occurrences actually containing data. However, it is possible to have blank occurrences preceding occurrences containing data.

This count is different from the maximum capacity of an array, which you can retrieve with `sm_max_occur`.

RETURNS

The highest numbered occurrence containing data.
0 if there is no data in the field.
–1 if the field is not found.

VARIANTS

```
sm_n_num_occurs(field_name);
```

EXAMPLE

```
#include "smdefs.h"

/* Compute the number of unused items in this
 * scrollable field. */

int maximum, used, unused;

maximum = sm_n_max_occur ("hatpins");
used = sm_n_num_occurs ("hatpins");
unused = maximum - used;
```

O_

variants that take a field number & occurrence number

SYNOPSIS

```
sm_o_...(field_number, occurrence, ...)
int field_number;
int occurrence;
```

DESCRIPTION

The `o_` functions refer to data by field number and occurrence number. An occurrence is a slot within an array of fields in which data may be stored. Occurrences may be either on or off–screen. Since **JAM** treats an individual field as an array with one field, even a single non–scrolling field is considered to have one occurrence. The **JAM** library contains routines that allow you to manipulate individual occurrences during run–time.

If `occurrence` is zero, the reference is always to the current contents of the specified field, or if the field is an array, to the current contents of the base field of the array.

For the description of a particular function, look under the related function without `o_` in its name. For example, `sm_o_amt_format` is described under `sm_amt_format`.

occur_no

get the current occurrence number

SYNOPSIS

```
int sm_occur_no();
```

DESCRIPTION

This function returns the occurrence number of the field beneath the cursor. If the field is an element of a non–scrollable array, the occurrence number is the same as the field's element number. Likewise, the occurrence number of a single non–scrolling field is 1.

RETURNS

0 if the cursor is not in a field.
Otherwise, the occurrence number.

RELATED FUNCTIONS

```
sm_getcurno();
```

EXAMPLE

```
#include "smdefs.h"

/* Find the occurrence number of the field under the*/
/* cursor, and scroll down to the next higher*/
/* multiple of 5. */

int thisn;

thisn = sm_occur_no ();
sm_rscroll (sm_getcurno (), 5 - (thisn % 5));
```

off_gofield

move the cursor into a field, offset from the left

SYNOPSIS

```
int sm_off_gofield(field_number, offset)
int field_number;
int offset;
```

DESCRIPTION

This function moves the cursor into `field_number`, at position `offset` within the field's contents, regardless of the field's justification. The field's contents are shifted if necessary to bring the appropriate piece onscreen.

If `offset` is larger than the field length (or the maximum length if the field is shiftable), the cursor is placed in the rightmost position.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
sm_e_off_gofield(field_name, element, offset);
sm_i_off_gofield(field_name, occurrence, offset);
sm_n_off_gofield(field_name, offset);
sm_o_off_gofield(field_number, occurrence, offset);
```

RELATED FUNCTIONS

```
sm_disp_off();
sm_gofield(field_number);
sm_sh_off();
```

EXAMPLE

```
#include "smdefs.h"
#include <ctype.h>
/* Place cursor over the first embedded blank in the "names" field. */

char buf[256], *p;
int length;

length = sm_n_getfield (buf, "names");
for (p = buf; p <buf + length; ++p)
{
    if (isspace (*p))
        break;
}
sm_n_off_gofield ("names", p - buf);
```

option

set a Screen Manager option

SYNOPSIS

```
int sm_option(option, newval)
int option;
int newval;
```

DESCRIPTION

Use `sm_option` to alter during run–time the default Screen Manager options defined in `smsetup.h`. Possible options include, error window attributes, delayed write options, cursor display and zoom options. See the "Setup File" section in the *Configuration Guide* for a list of options and possible values. Use `sm_keyoption` to alter the behavior of cursor control keys.

If you wish to simply inquire as to an option's current value, use the value NOCHANGE (defined in `smsetup.h`) for `newval`.

This function replaces the following version 4.0 functions: `sm_ch_emsgatt`, `sm_ch_form_atts`, `sm_ch_qmsgatt`, `sm_ch_umsgatt`, `sm_dw_options`, `sm_er_options`, `sm_fcase`, `sm_fextension`, `sm_ind_set`, `sm_mp_options`, `sm_mp_string`, `sm_ok_options`, `sm_stextatt`, and `sm_zm_options`. They are included in your version 5.0 library only for backward compatibility. We strongly recommend that you do not use them in the future.

RETURNS

The old value for the specified option.
–1 if the option is out of range.

RELATED FUNCTIONS

```
sm_keyoption(key, mode, newval);
```

EXAMPLE

```
/* Put ^pulldown_leave in a control string to leave submenus open. */
/* or ^pulldown_close to make them close. */
```

```
/* Leave submenu open when choice is made. */
int
pulldown_leave(ignored_data)
char *ignored_data;
{
    sm_option(IN_SUBMENU,OK_LEAVEOPEN);
    return(0);
}


/* Close submenu when choice is made. */
int
pulldown_close(ignored_data)
char *ignored_data;
{
    sm_option(IN_SUBMENU,OK_CLOSE);
}
```

oshift

shift a field by a given amount

SYNOPSIS

```
int sm_oshift(field_number, offset)
int field_number;
int offset;
```

DESCRIPTION

This function shifts the contents of `field_number` by `offset` positions. If `offset` is negative, the contents are shifted right (data past the left-hand edge of the field become visible); otherwise, the contents are shifted left. Shifting indicators, if displayed, are adjusted accordingly.

The field may be shifted by fewer than `offset` positions if the maximum shifting width is reached with less shifting.

RETURNS

The number of positions actually shifted.
0 if the field is not found or is not shifting.

VARIANTS

```
sm_n_oshift(field_name, offset);
```

EXAMPLE

```
#include "smdefs.h"

/* Shift the Republicans gently toward the left,
 * and the Democrats toward the right.
 * For extra credit, speculate on which shift
 * is positive. */

sm_n_oshift ("REP", 1);
sm_n_oshift ("DEM", -1);
```

pinquire

obtain value of a global string

SYNOPSIS

```
#include "smglobs.h"

char *sm_pinquire(which)
int which;
```

DESCRIPTION

This function is used to obtain the current value of a global pointer variable. The mnemonics for `which` are defined in `smglobs.h`. If you wish to modify a global string use `sm_pset`.

Pointer values for `which` are defined in `smglobs.h`. They are:

Mnemonic	*Meaning*
P_YES	The Y character for YES/NO field. This is returned as a three character string. The first character is the lowercase yes value, the second character is the uppercase yes value, and the third character is the null terminator.
P_NO	The N character for YES/NO field. This is returned as a three character string. The first character is the lowercase no value, the second character is the uppercase no value, and the third character is the null terminator.
P_DECIMAL	This is returned as a three character string. The first character is the user's decimal point marker, the second character is the operating system's decimal point marker, and the third character is the null terminator.
P_FLDPTRS	Pointer to an array of field structures. The implementation of these structures is very release dependent.
P_TERM	Returns the name JAM uses as the terminal identifier or the null string if not found.
P_SPMASK	Pointer to a memory–resident, full–size form containing all blanks.

Mnemonic	*Meaning*
P_USER	Pointer to developer-specified region of memory. This pointer is not set by JAM; it is set and maintained, if desired, by the application.
SP_NAME	Name of the active screen.
SP_STATLINE	Text of current status line.
SP_STATATTR	Attributes of current status line (pointer to array of unsigned short integers).
P_DICNAME	Name of data dictionary file.
V_	Any of the "V_" mnemonics defined in `smvideo.h` may be passed to obtain various video related information.

In general, the objects pointed to by the pointers returned by `sm_pinquire` have limited duration and should be used or copied quickly (except for P_USER, which is maintained by the application). The P_ pointers point to the actual objects within JAM. The SP_ pointers point to copies of the objects. Since the characteristics of these objects are implementation dependent, they may change in future releases of JAM. In no case (except P_USER) should the objects be modified directly through the pointers returned by `sm_pinquire`. Use `sm_pset` to modify selected objects).

RETURNS

If the argument corresponds to a global pointer variable, the value of that variable is returned.
0 otherwise.

RELATED FUNCTIONS

```
sm_finquire(field_number, which);
sm_gp_inquire(group_name, which);
sm_iset(which, newval);
sm_pset(which, newval);
```

EXAMPLE

```
/* Get next key from user. Return -1 for 'n', 1 for 'y', and 0 */
/* if we don't know. 'n' and 'y' come from the message file, and */
/* so can be changed to reflect the local language. */
int
get_yes_no()
{
```

```
    unsigned key;
    char *yes;
    char *no;
    key = sm_getkey();
    yes = sm_pinquire(P_YES);
    no = sm_pinquire(P_NO);
    if (key == yes[0] || key == yes[1])
        return(1);
    if (key == no[0] || key == no[1])
        return(-1);
    return(0);
}
```

protect

protect an array

SYNOPSIS

```
int sm_aprotect(field_number, mask)
int sm_aunprotect(field_number, mask)
int sm_protect(field_number)
int sm_unprotect(field_number)
int sm_1protect(field_number, mask)
int sm_1unprotect(field_number, mask)

int field_number;
int mask;
```

DESCRIPTION

There are four types of protection associated with fields and arrays, any combination of which may be assigned: data entry, tabbing into, clearing, and validation. `sm_protect` and `sm_unprotect` always set and clear all four types of protection. The remaining protection functions set and clear any combination of protection, as specified by mask. The mnemonics for mask are defined in `smflags.hsmvalids.h` and are listed below. Combinations may be specified by oring mnemonics together.

Mnemonic for mask	*Meaning*
EPROTECT	protect from data entry
TPROTECT	protect from tabbing into and from entering via any other key
CPROTECT	protect from clearing
VPROTECT	protect from validation
ALLPROTECT	protect from all of the above

Protection is associated with an individual field (i.e. an element), and with an array as a whole. Therefore, all offscreen array occurrences always share the same level of protection, while the onscreen occurrences have the levels of protection (possibly all different) associated with their host fields (i.e. elements). Since protection is associated with individual fields, and not with individual occurrences, deleting an occurrence with `sm_doccur` does not scroll up the protection with the occurrences.

sm_protect, sm_unprotect, sm_1protect, and sm_1unprotect set and clear protection for individual fields. sm_aprotect and sm_aunprotect set and clear protection for all of the fields of an array, and for the array as a whole (the field_number may specify any field in the array). For example, unprotecting an array with sm_aunprotect undoes protection done by sm_1protect. A subsequent call to sm_1protect re-protects the specified field of the array, but never affects the offscreen occurrences of the array.

Caution: It is generally safer to protect and unprotect arrays with sm_aprotect and sm_aunprotect, rather than with the field-oriented protection functions.

RETURNS

-1 if the field does not exist;
0 otherwise.

VARIANTS

```
sm_n_protect(field_name);
sm_e_protect(field_name, element);
sm_n_unprotect(field_name);
sm_e_unprotect(field_name, element);
sm_n_1protect(field_name, mask);
sm_e_1protect(field_name, element, mask);
sm_n_1unprotect(field_name, mask);
sm_e_1unprotect(field_name, element, mask);
sm_n_aprotect(field_name, mask);
sm_n_aunprotect(field_name, mask);
```

EXAMPLE

```
#include "smdefs.h"

/* Postpone calculations by protecting the subtotals
   column from validation. This will prevent execution
   of its math edit. */
sm_n_aprotect("subtotals", VPROTECT);

/* Protect the partnum array from data entry and clearing,
 * while still allowing the cursor to enter it. This
 * allows the user to scroll through partnum and select the
 * desired partnum. */
sm_n_aprotect("partnum", EPROTECT | CPROTECT);
```

pset

Modify value of global strings

SYNOPSIS

```
#include "smglobs.h"

char *sm_pset(which, newval)
int which;
char *newval;
```

DESCRIPTION

This function is used to modify the contents of a global string. The string you wish to change is specified by `which`. The value that you wish to change the variable to is specified by `newval`. If you wish only to get the value of a global string use `sm_pinquire`.

The following values for `which`, defined in `smglobs.h`, are available:

Mnemonic	*Meaning*
P_YES	The Y character for YES/NO field. This is specified by a three character string. The first character is the lowercase yes value, the second character is the uppercase yes value, and the third character is the null terminator.
P_NO	The N character for YES/NO field. This is specified by a three character string. The first character is the lowercase no value, the second character is the uppercase no value, and the third character is the null terminator.
P_DECIMAL	This is specified by a three character string. The first character is the user's decimal point marker, the second character is the operating system's decimal point marker, and the third character is the null terminator.
P_USER	Pointer to developer–specified region of memory. This pointer is not set by JAM; it is set and maintained, if desired, by the application.
SP_NAME	Name of the active screen.

Mnemonic	*Meaning*
SP_STATLINE	Text of current status line.
SP_STATATTR	Attributes of current status line (pointer to array of unsigned short integers).
V_	Any of the "V_" mnemonics defined in `smvideo.h` may be specified to change video related information.

RETURNS

If `which` is one of the above, the old contents of the corresponding array are copied into
a buffer, and a pointer to that buffer is returned.
0 otherwise.

RELATED FUNCTIONS

```
sm_iset(which, newval);
sm_pinquire(which);
```

EXAMPLE

```
/* Set things for "german":   Ja == yes, */
/* Nein == no, and ',' is decimal point. */

void
set_german()
{
    sm_pset(P_YES,"jJ");
    sm_pset(P_NO,"nN");
    sm_pset(P_DECIMAL,",.");
    sm_err_reset("Jetzt spreche ich Deutsch!");
}
```

putfield

put a string into a field

SYNOPSIS

```
int sm_putfield(field_number, data)
int field_number;
char *data;
```

DESCRIPTION

The string `data` is moved into the field specified by `field_number`. Strings that are too long are truncated without warning, while strings shorter than the destination field are blank filled (to the left if the field is right justified, otherwise to the right). If `data` is a null string, then the field is cleared. This causes date and time fields that take system values to be refreshed.

This function sets the field's MDT bit to indicate that it has been modified, and clears its VALIDED bit to indicate that the field must be revalidated upon exit. `sm_n_putfield` and `sm_i_putfield` store `data` in the LDB if the named field is not present in the screen. However, if the LDB item has a scope of 1 (constant), its contents are unaltered and the function returns –1.

In variants that take `name` as an argument, `name` can be either the name of a field or a group. In the case of a group, the functions `sm_select` and `sm_deselect` should be used to change the group's value.

Notice that the order of arguments to this function is different from that of arguments to the related function `sm_getfield`.

RETURNS

–1 if the field is not found; 0 otherwise.

VARIANTS

```
sm_e_putfield(name, element, data);
sm_i_putfield(name, occurrence, data);
sm_n_putfield(name, data);
sm_o_putfield(field_number, occurrence, data);
```

RELATED FUNCTIONS

```
sm_deselect(group_name, group_occurrence);
sm_getfield(buffer, field_number);
sm_select(group_name, group_occurrence);
```

EXAMPLE

```
#include "smdefs.h"

sm_putfield (1, "This string has 29 characters");
```

putjctrl

associate a control string with a key

SYNOPSIS

```
#include "smkeys.h"

int sm_putjctrl(key, control_string, scope)
int key;
char *control_string;
int scope;
```

DESCRIPTION

Each JAM screen contains a table of control strings associated with function keys, the PF, SPF, and APP keys. JAM also maintains a default table of keys and control strings, which take effect when the current screen has no control string for a function key you press. This table enables you to define system–wide actions for keys. It is initialized from the SMINICTRL setup variables. See the section on setup in the *Configuration Guide* for further information.

This function associates `control_string` with `key` in one of the tables, replacing the control string previously associated with `key` (if there was one). If `scope` is zero, the control string is installed in the current screen, and disappears when you exit the screen; otherwise, it goes into the system–wide default table. If `control_string` is empty, the existing control string, if any, is deleted. If both screen and default control strings exist for a given key, deleting the control string for the screen puts the default control string into effect.

If you install a default control string for a key that is defined in the current screen, the definition in the screen is used. Note also that JAM does not search the form or window stack for function key definitions; only the current screen and the default table are consulted. Mnemonics for `key` are in `smkeys.h`. The syntax for control strings is defined in the *Author's Guide*.

RETURNS

–5 if insufficient memory is available; 0 otherwise.

EXAMPLE

```
#include "smkeys.h"

/* These 3 calls duplicate the defaults for the JAM run-time system.*/
sm_putjctrl (SPF1, "^jm_gotop", 1);
sm_putjctrl (SPF2, "^jm_system", 1);
sm_putjctrl (SPF3, "^jm_goform", 1);
```

pwrap

put text to a wordwrap field

SYNOPSIS

```
int sm_pwrap(field_number, text)
int field_number;
char *text;
```

DESCRIPTION

This function copies `text` to a wordwrap field specified by `field_number`. Wraps occur at the end of words. The last character of every line is a space. If a word is longer than one less than the length of the field, the word is broken one character short of the end of the field, a space is appended, and the remainder of the word wraps to the next line.

The variant `sm_o_pwrap` copies the text into an array beginning at the specified occurrence.

Warning: If you attempt to copy data that is too large for the wordwrap field to hold, `sm_pwrap` truncates the excess text.

RETURNS

–1 if the field number is invalid.
–2 if the text was truncated because it was too long for the field.
0 otherwise.

VARIANTS

```
sm_o_pwrap(field_number, occurrence, text);
```

RELATED FUNCTIONS

```
sm_gwrap(buffer, field_number, buffer_length);
```

query_msg

display a question, and return a yes or no answer

SYNOPSIS

```
int sm_query_msg(message)
char *message;
```

DESCRIPTION

The `message` is displayed on the status line, until you type a yes or a no key. A yes key is the first letter of the `SM_YES` entry in the message file (or the XMIT key), and a no key is the first letter of the `SM_NO` entry (or the EXIT key); case is ignored. At that point, this function returns the lower case letter as defined in the message file to its caller.

All keys other than yes and no keys are ignored.

Several *percent escapes* provide control over the content and presentation of status messages. See `sm_emsg` for details.

RETURNS

Lower–case ASCII 'y' or 'n', according to the response.

RELATED FUNCTIONS

```
sm_d_msg_line(message, display_attribute);
sm_is_no(field_number);
sm_is_yes(field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Ask a couple of straightforward questions. Be careful
 * of the dangling else, which has ruined many
 * relationships. */

if (sm_query_msg("Are you single?") == 'y')
    if (sm_query_msg("Will you go out with me?") == 'y')
        if (sm_query_msg(
            "Do you like Clint Eastwood movies?")
            == 'n')
            ...
```

qui_msg

display a message preceded by a constant tag, and reset the status line

SYNOPSIS

```
void sm_qui_msg(message)
char *message;
```

DESCRIPTION

This function prepends a tag (normally "ERROR:") to `message`, and displays the whole on the status line (or in a window if it is too long). The tag may be altered by changing the `SM_ERROR` entry in the message file. The message remains visible until the operator presses a key. Refer to the description of setup in the *Configuration Guide* for an exact description of error message acknowledgement. If the message is longer than the status line, it is displayed in a window instead. If the cursor position display has been turned on (see `sm_c_vis`), the end of the status line contains the cursor's current row and column. If the message text would overlap that area of the status line, it is displayed in a window instead.

This function is identical to `sm_quiet_err`, except that it does not turn the cursor on. It is similar to `sm_emsg`, which does not prepend a tag.

Several *percent escapes* provide control over the content and presentation of status messages. See `sm_emsg` for details.

RELATED FUNCTIONS

```
sm_emsg(message);
sm_err_reset(message);
sm_option(option, newval);
sm_quiet_err(message);
```

EXAMPLE

```
#include "smdefs.h"

sm_qui_msg ("Be %A17veewwwwy%A7 quiet. "
    "I'm hunting wabbits.");
```

quiet_err

display error message preceded by a constant tag, and reset the status line

SYNOPSIS

```
void sm_quiet_err(message)
char *message;
```

DESCRIPTION

This function prepends a tag (normally "ERROR") to `message`, turns the cursor on, and displays the whole message on the status line (or in a window if it is too long). This function is identical to `sm_qui_msg`, except that it turns the cursor on. It is similar to `sm_err_reset`, which does not prepend a tag. Refer to `sm_emsg` for an explanation of how to change display attributes and insert function key names within a message.

RELATED FUNCTIONS

```
sm_emsg(message);
sm_err_reset(message);
sm_option(option, newval);
sm_qui_msg(message);
```

EXAMPLE

```
/* Display an error message that is surely long
 * enough to be put into a window. */

char *buf;

if ((buf = malloc (8192)) == 0)
{
    sm_quiet_err ("Sorry, guy, I'm %A0017all%A7 out "
    "of memory. Here's 500 bucks, why don't you just "
    "run down to the corner dealer and pick me up "
    "me up a meg?");
    sm_cancel ();
}
```

rd_part

read part of a data structure to the current screen

SYNOPSIS

```
void sm_rd_part(screen_struct, first_field, last_field,
      language)
char *screen_struct;
int first_field;
int last_field;
int language;
```

DESCRIPTION

This function copies data from a structure to all fields between `first_field` and `last_field` within the current screen, converting individual members as appropriate. An array and its scrolling occurrences are copied only if the *first* element falls between `first_field` and `last_field`. This routine is commonly used with `sm_wrt_part`, which writes part of the screen to a structure. If you wish to read information into the entire screen, use `sm_rdstruct`. To read information into a data dictionary record, use `sm_rrecord`. Use `sm_putfield` to write a string to an individual field.

The structure declaration can be automatically generated from a screen file with the utility `f2struct`. Each member of the structure is a field of the type specifed in the Screen Editor. If you specify the type `omit`, data is not written into the field. See "Data Type" in the *Author's Guide* and `f2struct` in the *Utilities Guide* for further information.

Once created, the declaration may be treated exactly like any other structure declaration. You can ignore the items that represent fields which do not fall within the bounds of the specifed fields. However, the structure definition must contain all of the fields on the screen. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`.

The arguments that represent the range of fields to be copied, `first_field` and `last_field` are passed as field numbers.

The argument `language` stands for the programming language in which the structure is defined. It controls the conversion of string and numeric data.

The following values for `language` are defined in `smumisc.h`:

Language	*Meaning*
S_C_NULL	C with null–terminated strings. This is the most common choice.
S_C_BLNK	C with blank–filled strings. Used for compactness and compatibility with other languages.

The structure may be initialized with `sm_wrt_part` or with data from elsewhere. Structure members within the specified range which will not be initialized prior to calling `sm_rd_part` must be zeroed–out or you risk crashing your application when garbage is read into the screen.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
sm_putfield(field_number, data);
sm_rd_struct(screen_struct, byte_count, language);
sm_rrecord(structure_ptr, record_name, byte_count, language);
sm_wrt_part(screen_struct, first_field, last_field, language);
```

EXAMPLE

```
Refer to sm_wrt_part for a rather lengthy example.
```

rdstruct

read data from a structure to the screen

SYNOPSIS

```
void sm_rdstruct(screen_struct, byte_count, language)
char *screen_struct;
int *byte_count;
int language;
```

DESCRIPTION

This function copies data from a structure to the current screen, converting individual members as appropriate. It is commonly used with `sm_wrtstruct`, which writes data from fields on the current screen to a structure. If you wish to read information into a group of consecutively numbered fields, use `sm_rd_part`. To read information from a data dictionary record, use `sm_rrecord`. Use `sm_putfield` to write a string to an individual field.

The structure declaration can be automatically generated from a screen file with the utility `f2struct`. Each member of the structure is a field of the type specifed in the Screen Editor. If you specify the type `omit`, data is not written into the field. See "Data Type" in the *Author's Guide* and `f2struct` in the *Utilities Guide* for further information.

Once created, the declaration may be treated exactly like any other structure declaration. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`.

The argument `byte_count` is the address of an integer variable. `sm_rdstruct` stores in `byte_count` the number of bytes copied from the structure.

The argument `language` stands for the programming language in which the structure is defined. It controls the conversion of string and numeric data. This must be consistent with how the structure was created with `f2struct`.

The following values for `language` are defined in `smumisc.h`:

Language	*Meaning*
S_C_NULL	C with null–terminated strings. This is the most common choice.
S_C_BLNK	C with blank–filled strings. Used for compactness and compatibility with other languages.

The structure may be initialized with `sm_wrtstruct` or with data from elsewhere. Members within the structure that will not be initialized prior to calling `sm_rdstruct` must be zeroed–out or you risk crashing your application when garbage is read into the screen.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
sm_putfield(field_number, data);
sm_rd_part(screen_struct, first_field, last_field, language);
sm_rrecord(structure_ptr, record_name, byte_count, language);
sm_wrtstruct(screen_struct, byte_count, language);
```

EXAMPLE

```
Please refer to sm_wrtstruct for an extended example.
```

rescreen

refresh the data displayed on the screen

SYNOPSIS

```
void sm_rescreen();
```

DESCRIPTION

This function repaints the entire display from JAM's internal screen and attribute buffers. Anything written to the screen by means other than JAM library functions are erased. This function is normally bound to the RESCREEN key and executed automatically within `sm_getkey`.

You may need to use this function after doing screen I/O with the flag `sm_do_not_display` turned on, or after escaping from an JAM application to another program (see `sm_leave`). If all you want is to force writes to the display, use `sm_flush`.

RELATED FUNCTIONS

```
sm_flush();
sm_return();
```

EXAMPLE

```
/* Mess the screen up good and proper, then restore
 * it with a call to sm_rescreen. */

for (i=1; i<30; i++)
{
    printf("************************************");
    printf("************************************\n");
}

sm_rescreen();
sm_err_reset("Verify that the screen has been "
    "restored.");
```

resetcrt

reset the terminal to operating system default state

SYNOPSIS

```
void sm_resetcrt();
void sm_jresetcrt();
void sm_jxresetcrt();
```

DESCRIPTION

The function `sm_resetcrt` is generally used only when you are writing your own Executive. It resets terminal characteristics to the operating system's normal state. Be sure to call `sm_resetcrt` be called when leaving the Screen Manager environment (before program exit).

All the memory associated with the display and open screens is freed. However, the buffers holding the message file, key translation file, etc. are not released. A subsequent call to `sm_initcrt` will find them in place. Then `sm_resetcrt` clears the screen and turns on the cursor, transmits the `RESET` sequence defined in the video file, and resets the operating system channel.

The **JAM** Executive calls `sm_resetcrt` via `sm_jresetcrt` (or via `sm_jxresetcrt` in the case of an authoring executable) automatically as part of its exit processing. It should not be called by application programs except in case of abnormal termination.

RELATED FUNCTIONS

```
sm_cancel();
sm_leave();
```

EXAMPLE

```
/* If an effort to read the first form results in
 * failure, clean up the screen and leave. */

if (sm_r_form ("first") < 0)
{
    sm_resetcrt ();
    exit (1);
}
```

resize

notify **JAM** of a change in the display size

SYNOPSIS

```
int sm_resize(rows, columns)
int rows;
int columns;
```

DESCRIPTION

This function enables you to change the size of the display used by **JAM** from the default defined by the LINES and COLMS entries in the video file. It makes it possible to use a single video file in a windowing environment. Applications can be run in different sized windows with each application setting its display size at run time. It can also be used for switching between normal and compressed modes (e.g. 80 and 132 columns on VT100–compatible terminals).

If the specified rectangle is larger than the physical display, the results are unpredictable. You may specify at most 255 rows or columns.

RETURNS

–1 if a parameter was less than 0 or greater than 255.
0 if successful.
Program exit on memory allocation failure.

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"
#include "smglobs.h"
#define WIDTH_TOGGLE PF9

/* Somewhat irregular code to switch a VT-100
 * between 80- and 132-column mode by pressing PF9. */

switch (sm_input (IN_DATA))
{
...
case WIDTH_TOGGLE:
    if (sm_inquire(I_MXCOLMS) == 80)
    {
        printf ("\033[?3h");
        sm_resize (sm_inquire(I_MXLINES), 132);
    }
    else
```

```
    {
        printf ("\033[?31");
        sm_resize (sm_inquire(I_MXLINES), 80);
    }
    break;
...
}
```

restore_data

restore previously saved data to the screen

SYNOPSIS

```
int sm_restore_data(buffer)
char *buffer;
```

DESCRIPTION

This function restores all data items, both onscreen and offscreen, to the current screen from an area initialized by `sm_save_data`. `buffer` is the address of the area. Passing an address not returned by `sm_save_data`, or attempting to restore to a screen other than the one saved, can produce unpredictable results.

Data items are stored in the save–data buffer as null–terminated character strings. The contents of a scrollable array is preceded by 2 bytes giving the total number of items saved (high order byte first); each item is preceded by two bytes of display attribute, and followed by a null. There is an additional null following all the scrolling data.

RETURNS

–1 if an error occurred, usually memory allocation failure.
0 otherwise.

return

prepare for return to **JAM** application

SYNOPSIS

```
void sm_return();
```

DESCRIPTION

This routine should be called upon returning to a JAM application after a temporary exit.

It sets up the operating system channel and initializes the display using the SETUP string from the video file. It does *not* restore the screen to the state it was in before sm_leave was called. Use sm_rescreen to accomplish that, if desired.

RELATED FUNCTIONS

```
sm_leave();
sm_resetcrt();
```

EXAMPLE

```
#include "smdefs.h"

/* Escape to the UNIX shell for a directory listing */

sm_leave ();
system ("ls -l");
sm_return ();
sm_c_off ();
sm_d_msg_line ("Hit any key to continue",
                BLINK | WHITE);
sm_getkey ();
sm_d_msg_line ("", WHITE);
sm_rescreen ();
```

rmformlist

empty the memory–resident form list

SYNOPSIS

```
void sm_rmformlist();
```

DESCRIPTION

This function erases the memory–resident form list established by `sm_formlist`, and releases the memory used to hold it. It does not release any of the memory–resident JPL modules, key sets, or screens themselves. Calling this function prevents `sm_r_window`, `sm_r_keyset`, `sm_jplcall`, and related functions from finding memory–resident objects.

RELATED FUNCTIONS

```
sm_formlist(ptr_to_form_list);
```

EXAMPLE

```
/* Hide all the memory-resident screens, perhaps
 * because the disk versions have been modified. */

sm_rmformlist ();
```

rrecord

read data from a structure to a data dictionary record

SYNOPSIS

```
void sm_rrecord(structure_ptr, record_name, byte_count,
      language)
char *structure_ptr;
char *record_name;
int *byte_count;
int language;
```

DESCRIPTION

This function reads data from a C structure into fields on the current screen that are part of a common data dictionary record. If a field is not on the current screen then the data is written to the LDB. This routine is commonly used with `sm_wrecord`, which writes data from a data dictionary record to a C structure. If you wish to read data into all of the fields within the current screen, use `sm_rdstruct`. To copy data to a group of consecutively numbered fields, use `sm_rd_part`. Use `sm_putfield` to write a string to an individual field.

To automatically generate a file containing a structure declaration for each data dictionary record, use the `dd2struct` utility. Each structure member is a field within a data dictionary record that is of the type specified in the Screen Editor. Data is written into the field onscreen even if the `omit` type is specified. See "Data Type" in the *Author's Guide* and `dd2struct` in the *Utilities Guide* for further information.

Once created, the declarations may be treated exactly like any other structure declarations. The argument `struct_ptr` is the address of a variable of one of the structure types generated by `dd2struct`. The argument `record_name` is the name of the data dictionary record from which the structure was created.

The argument `byte_count` is a pointer to an integer. Upon return from `sm_rrecord`, the value contained in the integer is the number of bytes or characters read from the structure. The value is 0 if an error occurred.

The argument `language` stands for the programming language in which the structure is defined. it controls the conversion of string and numeric data.

The following values for `language` are defined in `smumisc.h`:

Language	*Meaning*
S_C_NULL	C with null–terminated strings. This is the most common choice.
S_C_BLNK	C with blank–filled strings. Used for compactness and compatibility with other languages.

The structure may be initialized with `sm_wrecord` or with data from elsewhere. Members within the structure that will not be initialized prior to calling `sm_rrecord` must be zeroed–out or you risk crashing your application when garbage is read into the screen or the LDB.

Remember, you must update the structure declaration whenever you alter the data dictionary from which it was generated.

RELATED FUNCTIONS

```
sm_putfield(field_number, data);
sm_rd_part(screen_struct, first_field, last_field, language);
sm_rd_struct(screen_struct, byte_count, language);
sm_wrecord(structure_ptr, record_name, byte_count, language);
```

rs_data

restore saved data to some of the screen

SYNOPSIS

```
void sm_rs_data(first_field, last_field, buffer)
int first_field;
int last_field;
char *buffer;
```

DESCRIPTION

All data items, both onscreen and offscreen, are restored to the fields between `first_field` and `last_field` from an area initialized by `sm_sv_data`. The address of the area is in `buffer`.

See `sm_sv_data` to create a buffer for subsequent retrieval by this function. If the range of fields passed to this function does not match that passed to `sm_sv_data`, or `buffer` is not a value returned by that function, grievous errors will probably occur.

The format of the data area is explained briefly under `sm_restore_data`.

RETURNS

–1 if an error occurred, usually memory allocation failure.
0 otherwise.

RELATED FUNCTIONS

`sm_sv_data(first_field, last_field)`

rscroll

scroll an array

SYNOPSIS

```
int sm_rscroll(field_number, req_scroll)
int field_number;
int req_scroll;
```

DESCRIPTION

This function scrolls an array along with any synchronized arrays by `req_scroll` occurrences. If `req_scroll` is positive, the array scrolls down (towards the bottom of the data); otherwise, it scrolls up.

The function returns the actual amount scrolled. This could be the amount requested, or a smaller value if the requested amount would bring the array past its beginning or end. If 0 is returned it means that the array was at its beginning or end, or an error occurred. Negative numbers indicate scrolling up occurred.

RETURNS

The actual amount scrolled. Positive numbers indicate downward scrolling while negative numbers mean upward scrolling.
0 if no scrolling or error.

VARIANTS

```
sm_n_rscroll(field_name, req_scroll);
```

RELATED FUNCTIONS

```
sm_ascroll(field_number, occurrence);
sm_t_scroll(field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Find the number of the scrolling item under
 * the cursor and scroll down to the next
 * higher multiple of 5. */

int thisn;

thisn = sm_occur_no ();
sm_rscroll (sm_getcurno (), 5 - (thisn % 5));
```

s_val

validate the current screen

SYNOPSIS

```
int sm_s_val();
```

DESCRIPTION

This function validates each field and occurrence, whether on or offscreen, that is not protected from validation (VPROTECT). It is called automatically from `sm_input` when the TRANSMIT key is hit while in data entry mode. `sm_s_val` also validates groups.

When the first element of a scrolling array is encountered, earlier offscreen occurrences are validated first. When the last element of a scrolling array is encountered, later offscreen occurrences are validated immediately after that element.

If synchronized arrays exist, the following occurs. When an offscreen occurrence is validated, the corresponding occurrences from synchronized arrays are validated as well. Synchronized array are validated in order according to their base field number. The offscreen occurrences *preceding* the synchronized arrays are validated before the first onscreen occurrence of the first (lowest base field number) of the synchronized arrays. Similarly, the offscreen occurrences *following* the arrays are validated immediately after the last onscreen occurrence of the last (highest base field number) array.

Validation	*Skip if valid*	*Skip if empty*
required	y	n
must fill	y	y
regular expression	y	y
range	y	y
check-digit	y	y
date or time	y	y
table lookup	y	y
currency format	y	n*

Validation	*Skip if valid*	*Skip if empty*
math expresssion	n	n
field validation	n	n
JPL function	n	n

* The currency format edit contains a skip–if–empty flag; see the *Author's Guide*.

If you need to force a skip–if–empty validation, make the field required. A field with embedded punctuation must contain at least one non–blank non–punctuation character in order to be considered non–empty; otherwise any non blank character makes the field non–empty.

If an occurrence fails validation, the cursor is positioned to it and an error message displayed. If the occurrence is offscreen, the array is scrolled to bring it onscreen. This routine returns at the first error; any fields past that error are not validated.

RETURNS

–1 if any field fails validation.
0 otherwise.

RELATED FUNCTIONS

```
sm_fval(field_number);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Treat the SPF1 key as transmit, for a change. */

int key;

sm_d_msg_line ("Press %KSPF1 when done.",
    WHITE | REVERSE);

while ((key = sm_input (IN_DATA)) != EXIT)
{
    if (key == SPF1)
    {
        if (sm_s_val ())
            sm_err_reset ("Please correct the "
                "mistake(s).");
        else
            break;
    }
}
...
```

save_data

save screen contents

SYNOPSIS

```
char *sm_save_data();
```

DESCRIPTION

The current screen's data is saved for external access or subsequent retrieval, and the address of the save area returned. Radio button and checklist selections are not saved.

To restore the saved data, use `sm_restore_data`. Refer to `sm_restore_data` function for a brief explanation of the save format. Use `sm_sv_free` to discard a save area.

You can get the size of the data with the following line of code:

```
length = ((unsigned int *)buffer)[-1];
```

RETURNS

0 if insufficient memory was available,
otherwise the address of a memory area containing the screen's data.

RELATED FUNCTIONS

```
sm_restore_data(buffer);
sm_sv_free(buffer)
```

sc_max

alter the maximum number of occurrences allowed in a scrollable array

SYNOPSIS

```
int sm_sc_max(field_number, new_max)
int field_number;
int new_max;
```

DESCRIPTION

This function changes the maximum number of occurrences allowed in `field_number`, and in all synchronized arrays. The original maximum is set when the screen is created. If the desired new maximum is less than the highest numbered occurrence that contains data, the new maximum is set to the number of that occurrence (i.e., the value returned by `sm_num_occurs`). The maximum can decrease only to a value between the highest numbered occurrence containing data and the previous maximum. It can never be less than the number of elements in the array.

RETURNS

The actual new maximum (see above).
0 if the desired maximum is invalid, or if the array is not scrollable.

VARIANTS

```
sm_n_sc_max(field_name, new_max);
```

RELATED FUNCTIONS

```
sm_max_occur(field_number);
sm_num_occurs(field_number);
```

EXAMPLE

```
#include "smdefs.h"
#define SCROLLNUM 7

/* When the number of occurrences entered in an array */
/* exceeds ten less than the maximum, increase */
/* the maximum by 100. */

int maxnow;

maxnow = sm_max_occurs (SCROLLNUM);
if (maxnow - sm_num_occurs (SCROLLNUM) < 10)
    sm_sc_max (SCROLLNUM, maxnow + 100);
```

sdtime

get formatted system date and time

SYNOPSIS

```
char *sm_sdtime(format)
char *format;
```

DESCRIPTION

This function gets the current date and/or time from the operating system and returns it in the form specified by `format`.

`format` is a string beginning with `y` or `n` followed by any combination of date/time tokens and literal text. `y` indicates a 12–hour clock; `n` (or any other character) indicates a 24–hour clock. This character must be given, even if the format does not include time tokens. The tokens are described in the table below. These tokens are case–sensitive.

Unit	*Description*	*Token*
Year	4 digit (e.g., 1990)	`%4y`
	2 digit (e.g., 90)	`%2y`
Month	1 or 2 digit (1 – 12)	`%m`
	2 digit (01 – 12)	`%0m`
	full name (e.g., January)	`%*m`
	3 character name (e.g., Jan)	`%3m`
Day	1 or 2 digit (1 – 31)	`%d`
	2 digit (01 – 31)	`%0d`
Day of the Week	full name (e.g. Sunday)	`%*d`
	3 character name (e.g., Sun)	`%3d`
Day of the Year	digit (1 – 365)	`%+d`

Unit	*Description*	*Token*
Hour	1 or 2 digit (1 – 12 or 1 – 24)	`%h`
	2 digit (01 –12 or 01 –24)	`%0h`
Minute	1 or 2 digit (1 – 59)	`%M`
	2 digit (01 – 59)	`%0M`
Second	1 or 2 digit (1 – 59)	`%s`
	2 digit (01 – 59)	`%0s`
AM or PM	for use with a 12–hour clock	`%p`
Literal Percent	use `%` as a literal character	`%%`
Ten Default Formats (from the message file; refer to the *Configuration Guide* for details.)	`SM_0DEF_DTIME`	`%0f`
	`SM_1DEF_DTIME`	`%1f`
	...	`...`
	`SM_9DEF_DTIME`	`%09f`

At runtime, JAM strips off the first character of `format`. If the character is `y`, it uses a 12–hour clock; else it uses the default 24–hour clock. Next it examines the rest of `format`, replacing any tokens with the appropriate values. All other characters are used literally. Therefore, be sure to put a `y` or an `n` (or perhaps a blank) at the beginning of `format`. If you do not, JAM strips off the first token's percent sign and it treats the rest of the token as literal text.

You may also retrieve a date/time format from a field using `sm_edit_ptr`.

The text for day and month names, AM and PM, as well as the tokens for the ten default formats, are all stored in the message file. These entries may be modified. See the *Configuration Guide* for details.

This function uses a static buffer which it shares with other date and time formatting functions. The buffer is 256 bytes long. There is no checking for overflow. You should process the returned string, or copy it to a local variable, before making additional function calls.

Note: This function replaces Release 4's `sm_sdate` and `sm_stime` function.

RETURNS

A pointer to the current date/time in the specified format.
Empty if `format` is invalid.

RELATED FUNCTIONS

```
sm_calc(field_number, occurrence, expression);
sm_udtime(time, format)
```

EXAMPLE

```
#include "smdefs.h"
/* Put the current date MONTH-DAY-YEAR in the field "time". */
char *format;
format = "n%m-%0d-%2y";
sm_n_putfield ("time", sm_sdtime (format));
```

select

select a checklist or radio button occurrence

SYNOPSIS

```
int sm_select(group_name, group_occurrence)
char *group_name;
int group_occurrence;
```

DESCRIPTION

This function allows you to select a specific occurrence within a checklist or radio button. The group name and occurrence number are used to reference the desired selection.

Use `sm_deselect` to deselect a checklist occurrence.

Selecting a radio button occurrence automatically causes the currently selected radio button to be deselected, because exactly one occurrence in a radio button group must be selected at all times. See the *Author's Guide* for a more detailed discussion of groups.

Use `sm_isselected` to check whether or not a particular radio button or checklist occurrence is currently selected.

RETURNS

–1 arguments do not reference a checklist or radio button occurrence.
0 occurrence not previously selected.
1 occurrence previously selected.

RELATED FUNCTIONS

```
sm_deselect(group_name, group_occurrence);
sm_isselected(group_name, group_occurrence);
```

set_injpl

allow C routines to access JPL variables & subroutines

SYNOPSIS

```
int sm_set_injpl(mode)
int mode;
```

DESCRIPTION

This function provides a C routine access to JPL variables and subroutines, which is particularly useful with **JAM/DB*i***, as it allows `dbi_dbms` and `dbi_sql` statements to access JPL variables.

Normally a C routine that is called from JPL, via the JPL `call` statement, does not have access to either the "automatic" variables of the caller (the JPL `proc`) or the "static" variables in the module of the caller. If this routine is called with a `mode` that is non–zero, the C function will have access to both JPL "automatic" and "static" variables. It will also have access to any `proc`'s in the current (the caller's) JPL module. Thus it is as if the C function is embedded bodily within the JPL procedure.

The `mode` remains in effect until the calling JPL procedure is returned to, or `sm_set_injpl` is called again with a `mode` of zero. This means that all subroutines of the C routine will also have access to the current JPL module's variables and procedures. Of course, if the C routine calls a JPL `proc` (e.g. via `sm_jplcall`), the new JPL `proc` *will not* have access to variables in the JPL `proc` that called the C routine.

NOTE: This function should be used with care. For example, since `sm_jwindow` is a C subroutine, it too will have access to the current module's JPL variables and `procs`. In addition any screen entry, exit or validation functions will also have access to these variables and procedures. This can cause some unintended consequences when, for example, a JPL routine opens a screen, and the new screen's entry function calls a JPL `proc`. The JPL processor will look first in the original screen's JPL module (the current module) for the procedure, before it looks in the new screen's JPL module. If it finds a procedure of the same name in the current module it will execute that procedure instead of the procedure in the new screen's JPL. The safest way to use this routine is to set `mode` to a non–zero value when you require access, and then reset it promptly.

Note also that JPL variables are accessible only indirectly by `sm_calc` and colon–expansion (and by **JAM/DB*i*** functions such as `dbi_dbms` and `dbi_sql`), but not by `putfield`, `getfield`, and the like.

RETURNS

The previous value of `mode`

setbkstat

set background text for status line

SYNOPSIS

```
void sm_setbkstat(message, display_attribute)
char *message;
int display_attribute;
```

DESCRIPTION

The `message` is saved, to be shown on the status line whenever there is no higher priority message to be displayed. The highest priority messages are those passed to `sm_d_msg_line`, `sm_err_reset`, `sm_quiet_err`, or `sm_query_msg`; the next highest are those attached to a field by means of the status text option (see the JAM *Author's Guide*). Background status text has lowest priority.

Possible values for the `display_attribute` argument are defined in the header file `smattrib.h`, as shown in the table below:

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Highlights		Background Highlights	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Colors		Background Colors	
BLACK	0000	B_BLACK	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic causes the attribute to default to black.

`sm_setstatus` sets the background status to an alternating ready/wait flag; you should turn that feature off before calling this routine.

Refer to `sm_d_msg_line` for an explanation of how to embed attribute changes and function key names into your message.

RELATED FUNCTIONS

```
sm_d_msg_line(message, display_attribute);
sm_setstatus(mode);
```

setstatus

turn alternating background status message on or off

SYNOPSIS

```
void sm_setstatus(mode)
int mode;
```

DESCRIPTION

If mode is non–zero, alternating status flags are turned on. After this call, one message (normally Ready) is displayed on the status line while JAM is waiting for input, and another (normally Wait) when it is not. If mode is zero, the messages are turned off.

The status flags are replaced temporarily by messages passed to sm_err_reset or a related routine. They overwrite messages posted with sm_d_msg_line or sm_setbkstat.

The alternating messages are stored in the message file as SM_READY and SM_WAIT, and can be changed there. Attribute changes and function key names can be embedded in the messages; refer to sm_d_msg_line for instructions.

RELATED FUNCTIONS

```
sm_setbkstat(message, display_attribute);
```

EXAMPLE

```
#include "smdefs.h"
#include "smerror.h"
#define PAUSE (sm_flush (), sleep (3))
char buf[100];

/* Tell people what you're gonna tell 'em. */
sprintf (buf, "You will soon see %s alternating "
    "with %s below.",
    sm_msg_get (SM_READY), sm_msg_get (SM_WAIT));
sm_do_region (3, 0, 80, WHITE, buf);

/* Now tell 'em. */
sm_setstatus (1);
PAUSE;          /* Shows WAIT */
sm_input (IN_DATA);     /* Shows READY */

/* Finally, tell 'em what you told 'em. */
sprintf (buf, "That was %s alternating with %s "
    "on the status line.",
    sm_msg_get (SM_READY), sm_msg_get (SM_WAIT));
sm_err_reset (buf);
```

sh_off

determine the cursor location relative to the start of a shifting field

SYNOPSIS

```
int sm_sh_off();
```

DESCRIPTION

Returns the difference between the start of data in a shiftable field and the current cursor location. If the current field is not shiftable, it returns the difference between the leftmost column of the field and the current cursor location, like `sm_disp_off`.

RETURNS

The difference between the current cursor position and the start of shiftable data in the current field.
–1 if the cursor is not in a field.

RELATED FUNCTIONS

```
sm_disp_off();
```

EXAMPLE

```
#include "smdefs.h"

/* Fancy test to see whether the current field is shifted
 * to the left. */

if (sm_sh_off () != sm_disp_off ())
    sm_err_reset ("Ha! You shifted!");
```

shrink_to_fit

remove trailing empty array elements and shrink screen

SYNOPSIS

```
void sm_shrink_to_fit();
```

DESCRIPTION

Use this routine to dynamically downsize the current screen when you don't know how many elements of an array are going to be populated with data at run time. This routine removes the trailing elements in all arrays on a screen and then shrinks the screen to a size just large enough to accommodate the displayed data. If there is no data in the array, then the entire array is removed. Only the currently displayed copy of the screen in memory is altered.

The algorithm used in this function is designed to minimize screen size, but it never removes the first or last line of a screen. Therefore, in some cases, such as a 5 line screen containing a 5 element array in which 4 elements are populated, `sm_shrink_to_fit` does not shrink the array, since doing so would not shrink the screen.

This routine only downsizes the array and screen. It does not enlarge an array or screen that is too small to hold the information, so be sure to create, within the Screen Editor, an array and screen that can hold the largest amount of data that you plan on inserting.

EXAMPLE

```
/* Put ^shrink in the auto control */
/* to have window shrink to fit before */
/* user gets a chance to see it! */

int
shrink (ignored_data)
char *ignored_data;
{
    sm_shrink_to_fit();
    return (0);
}
```

sibling

define the current window as a sibling or not a sibling

SYNOPSIS

```
void sm_sibling(should_it_be)
int should_it_be;
```

DESCRIPTION

Users may switch between the active window and all siblings of that window while they are in viewport mode. Sibling windows must be next to each other on the window stack. When a window is defined as a sibling, then it and the window immediately beneath it on the window stack are considered to be siblings of one another. The user enters viewport mode when either the VWPT (viewport) logical key is pressed or when the application program makes a call to `sm_winsize`.

Use this function to define whether or not the current window is defined as sibling. To change the current sibling status of a window assign `should_it_be` to:

0	No, it is not a sibling window.
1	Yes, it is a sibling window.

To understand how sibling windows work, imagine you have a stack of three windows: `window_top`, `window_middle`, and `window_bottom`. To make `window_top` and `window_middle` siblings of each other, define `window_top` as a sibling window. They are now considered siblings of each other. You can then add a third sibling to the pair, by defining `window_middle` as a sibling window. This results in `window_middle` and `window_bottom` becoming siblings of one another and consequently, `window_top` and `window_bottom` are also siblings of each other. There is no limit to the number of siblings window you may chain together in this fashion, as long as the windows are adjacent to each other on the stack.

If you wish to bring a different window to the top of the stack, use `sm_wselect`. To get the number of windows currently in the window stack use `sm_wcount`.

The base form can be a sibling of the windows adjacent to it.

RELATED FUNCTIONS

```
sm_wcount();
sm_winsize();
sm_wrotate(step);
sm_wselect(window_number);
```

size_of_array

get the number of elements

SYNOPSIS

```
int sm_size_of_array(field_number)
int field_number;
```

DESCRIPTION

This function returns the number of elements in the array containing `field_number`. Elements are the onscreen portion of an array. An array always has at least one element.

RETURNS

0 if the field designation is invalid.
1 if the field is not an array.
The number of elements in the array otherwise.

VARIANTS

```
sm_n_size_of_array(field_name);
```

RELATED FUNCTIONS

```
sm_max_occur(field_number);
```

EXAMPLE

```
#define THEFIELD 6

/* Compute the number of pages of data in a
 * scrolling array, where a page is one
 * onscreen-array-full. */

int pages, elements;

elements = sm_size_of_array (THEFIELD);
pages = (sm_num_occurs (THEFIELD) + elements - 1)
    / elements;
```

skinq

obtain soft key information by position

SYNOPSIS

```
#include "smsoftk.h"

#include "smkeys.h"

int sm_skinq(scope, row, softkey, value, display_attribute,
      label1, label2)
int scope;
int row;
int softkey;
int *value;
int *display_attribute;
char *label1;
char *label2;
```

DESCRIPTION

Use this routine to obtain the value, attributes, and label of a soft key contained in a keyset currently in memory, given a soft key's position within a keyset.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Mnemonics for `scope` are defined in `smsoftk.h`. For a more detailed explanation of scope see the Keyset chapter of the *Programmer's Guide*.

The logical value of the specified soft key is placed in `value`. This will be a number that corresponds to a mnemonic defined in `smkeys.h`. A value of 0 means the key is inactive.

The attributes (color, blinking etc . . .) of the label are placed in `display_attribute`. The attribute should be one of the mnemonics listed in `smattrib.h`.

The first and second row labels are placed in `label1` and `label2` respectively. You should pre–allocate at least nine elements for `label1` and `label2` buffers (eight for the label characters and one for the null character).

If you do not desire information about one or more of these parameters you may assign the parameters the null pointer.

If you want general information about a keyset, see `sm_ksinq`. If you want the scope of the current keyset, use `sm_kscscope`.

WARNING: This routine cannot be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs JAM automatically breaks the rows to position them correctly on the monitor. This means that you cannot reliably reference a particular soft key by its row and position. Instead, use `sm_skvinq`.

RETURNS

0 if information has been returned.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
sm_kscscope();
sm_ksinq(scope, number_keys, number_rows, current_row,
      maximum_len, keyset_name);
sm_skvinq(scope, value, occurrence, attribute, label1, label2);
```

skmark

mark or unmark a soft key label by position

SYNOPSIS

```
#include "smsoftk.h"

int sm_skmark(scope, row, softkey, mark)
int scope;
int row;
int softkey;
int mark;
```

DESCRIPTION

Use this routine to mark or unmark a soft key label in an open keyset. The mark is made in the last position of the first label.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.h`. The argument `row` is the row number in which the desired `softkey` resides. Rows are counted from top to bottom, beginning with `1`. The argument `softkey` is the position number within `row` of the desired soft key. Positions are numbered left to right, beginning with `1`.

The argument `mark` may be any single ASCII character. An asterisk (*) is the most commonly used mark. To unmark the key use the space character (' ') for `mark`.

The marking or unmarking of a soft key is often done to indicate a selection on a function key that toggles between two options.

WARNING: This routine cannot be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs JAM automatically breaks the rows to position them correctly on the monitor. This means that you cannot reliably reference a particular soft key by its row and position. Instead, use `sm_skvmark`.

RETURNS

0 if the marking was successful.
–1 if there is no keyset of the specified scope.
–2 if the scope is out of range.
–3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
sm_skvmark(scope, value, occurrence, mark);
```

skset

set characteristics of a soft key by position

SYNOPSIS

```
#include "smsoftk.h"

#include "smkeys.h"

#include "smkeys.h"

int sm_skset(scope, row, softkey, value, attribute, label1,
      label2)
int scope;
int row;
int softkey;
int value;
int attribute;
char *label1;
char *label2;
```

DESCRIPTION

This routine can be used to modify a soft key's scope, value, attribute, or label of any currently open keysets. You may modify one or more of these specifications with each call of `sm_skset`.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.h`. The argument `row` is the row number in which the desired `softkey` resides. Rows are counted from top to bottom, beginning with `1`. The argument `softkey` is the position number within `row` of the desired soft key. Positions are numbered left to right, beginning with `1`.

The `value` refers to the logical key name to be assigned to the soft key. Available mnemonics are defined in `smkeys.h`. If you do not want to change the logical name, assign `-1` to `value`.

The `attribute` (color, blinking, etc.) is specified by using mnemonics listed in `smattrib.h`. If you do not want to change `attribute`, assign it `0`. (Note: If you set both the background and foreground to black, `sm_skset` sets the foreground to white, provided that the terminal supports background color.)

The variables `label1` and `label2` are the first and second lines of the labels respectively. If you do not wish to change one of the labels, assign it the null pointer.

WARNING: This routine cannot be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs JAM automatically breaks the rows to position them correctly on the monitor. This means that you cannot reliably reference a particular soft key by its row and position. Instead, use `sm_skvset`.

RETURNS

0 if no error has occurred.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
sm_skvset(scope, value, occurrence, newval, attribute, label1,
        label2);
```

skvinq

obtain soft key information by value

SYNOPSIS

```
#include "smsoftk.h"

#include "smkeys.h"

int sm_skvinq(scope, value, occurrence, attribute, label1,
      label2)
int scope;
int value;
int occurrence;
int *attribute;
char *label1;
char *label2;
```

DESCRIPTION

Use this routine to obtain the label text and attributes of a soft key contained in a keyset currently in memory, given the soft key's value. It can be used when the terminal has a different number of keys than the keyset was designed for.

The soft key is referenced by the keyset it belongs to, its value, and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.h`. The `value` of the soft key is one of the mnemonics defined in `smkeys.h`. The argument `occurrence` specifies which occurrence of a key with the specified value is desired (in case of duplicates).

The attributes (color, blinking etc . . .) of the label are placed in `attribute`. The value of the attributes correspond to a mnemonic, or some combination of ored mnemonics listed in `smattrib.h`.

The first and second row labels are placed in `label1` and `label2` respectively. You should pre–allocate at least nine elements for `label1` and `label2` buffers (eight for the label characters and one for the null character).

If you do not desire information about one or more of these parameters you may assign the parameters the null pointer.

For general information about a keyset, see `sm_ksinq`. If you want the scope of the current keyset, use `sm_kscscope`.

RETURNS

0 if information has been returned.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
sm_skinq(scope, row, softkey, value, display_attribute, label1,
      label2);
```

skvmark

mark a soft key by value

SYNOPSIS

```
#include "smsoftk.h"

#include "smkeys.h"

int sm_skvmark(scope, value, occurrence, mark)
int scope;
int value;
int occurrence;
int mark;
```

DESCRIPTION

Use this routine to mark or unmark a soft key label in an open keyset. The mark is made in the last position of the first label.

The soft key is referenced by the keyset it belongs to, its value and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.h`. The `value` of the soft key is one of the mnemonics defined in `smkeys.h`. The argument `occurrence` is the nth time that `value` appears in the keyset. If you wish to mark all occurrences of `value` assign 0 to `occurrence`.

The argument `mark` may be any single ASCII character. An asterisks (*) is the most commonly used mark. To unmark the key use the space character (' ') for `mark`.

The marking or unmarking of a soft key is often done to indicate a selection on a function key that toggles between two options.

RETURNS

0 if the mark was successful.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
sm_skmark(scope, row, softkey, mark);
```

skvset

set characteristics of a soft key by value

SYNOPSIS

```
#include "smsoftk.h"

#include "smkeys.h"

int sm_skvset(scope, value, occurrence, newval, attribute,
      label1, label2)
int scope;
int value;
int occurrence;
int newval;
int attribute;
char *label1;
char *label2;
```

DESCRIPTION

This routine can be used to modify the scope, value, attribute, or label of a soft key within a currently open keyset. You may modify one or more of these specifications with each call of `sm_skset`.

The soft key is referenced by the keyset it belongs to, its value and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.h`. The `value` of the soft key is one of the mnemonics defined in `smkeys.h`. The argument `occurrence` is the nth time that `value` appears in the keyset. If you wish to change all occurrences of `value` assign `0` to `occurrence`.

The value of `newvalue` refers to the logical key name to be assigned to the soft key. Available mnemonics are defined in `smkeys.h`. If you do want to change the logical name, assign `-1` to `value`.

The `attribute` (color, blinking, etc.) is specified by using mnemonics listed in `smattrib.h`. If you do not want to change `attribute`, assign it `0`. (Note: If you set both the background and foreground to black, `sm_skset` sets the foreground to white, provided that the terminal supports background color.)

The variables `label1` and `label2` are the first and second lines of the labels respectively. If you do not wish to change one of the labels, assign it the null pointer.

RETURNS

0 if no error occurred
–1 if there is no active keyset for the given scope

-2 for an invalid scope
-3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
sm_skset(scope, row, softkey, value, attribute, label1, label2);
```

soption

set a string option

SYNOPSIS

```
char *sm_soption(option, newval)
int option;
char *newval;
```

DESCRIPTION

Use `sm_soption` to alter during run-time the default string options defined in `smsetup.h`. The following table lists the valid mnemonics for `option`:

Mnemonic	*Description*
SO_EDITOR	Editor to use in JPL windows.
SO_FEXTENSION	Screen file extension.
SO_LPRINT	Operating system print command.
SO_PATH	Search path for screens and JPL procedures.

These variables are fully documented in the JAM *Configuration Guide*, under "System Environment and Setup Files."

RETURNS

The old value for the specified option.
0 if the option is invalid or a malloc error occurred.

RELATED FUNCTIONS

```
sm_option(option, newval);
```

EXAMPLE

```
char *default_lp;
default_lp = sm_soption (SMLPRINT, "lp -dny %s");
```

strip_amt_ptr

strip amount editing characters from a string

SYNOPSIS

```
char *sm_strip_amt_ptr(field_number, inbuf)
int field_number;
char *inbuf;
```

DESCRIPTION

Strips all non–digit characters from the string, except for an optional leading minus sign and decimal point. If `inbuf` is nonzero, `field_number` is ignored and the passed string is processed in place.

If `inbuf` is zero, the contents of `field_number` are used.

This function shares with several others a pool of buffers where it stores returned data. The value returned by any of them should therefore be processed quickly or copied. The shared pool is only used if `inbuf` is zero.

RETURNS

A pointer to a buffer containing the stripped text,
or 0 if `inbuf` is 0 and the field number is invalid.

RELATED FUNCTIONS

```
sm_amt_format(field_number, buffer);
sm_dblval(field_number);
```

EXAMPLE

```
#include "smdefs.h"

char *strip_text;
in amount;

strip_text = sm_strip_amt_ptr (0, "$1,234");
amount = atoi(strip_text);
```

submenu_close

close the current submenu

SYNOPSIS

```
int sm_submenu_close();
```

DESCRIPTION

Submenus are ordinarily closed before `sm_input` returns. It may, however, be told to leave them open by using the OK_LEAVEOPEN option, either in the setup file or via `sm_option`. See the *Configuration Guide* for details. Regardless of how this option is set, submenus are automatically closed whenever the underlying window is closed with `sm_close_window`.

This function, then, is needed only when all of the following conditions are true.

1. OK_LEAVEOPEN is in use.
2. The submenu is no longer needed.
3. Access is needed to the underlying window.

RETURNS

-1 if there is no submenu currently open.
0 otherwise.

RELATED FUNCTIONS

```
sm_close_window();
```

sv_data

save partial screen contents

SYNOPSIS

```
char *sm_sv_data(first_field, last_field)
int first_field;
int last_field;
```

DESCRIPTION

The current screen's data, from all fields numbered from `first_field` to `last_field`, is saved for external access or subsequent retrieval, and the address of the save area returned. Use `sm_rs_data` to restore it.

See `sm_save_data` for the save format.

RETURNS

The address of an area containing the saved data.
0 if the current screen has no fields, or sufficient free memory is not available.

RELATED FUNCTIONS

```
sm_rs_data(first_field, last_field, buffer)
sm_save_data();
sm_sv_free(buffer)
```

sv_free

free a save–data buffer

SYNOPSIS

```
void sm_sv_free(buffer)
char *buffer;
```

DESCRIPTION

The save area at `buffer`, which must have been created by `sm_save_data` or `sm_sv_data`, is released and is no longer accessible.

`sm_save_data` and related functions record the addresses of save areas, and after ten have been accumulated the oldest is released automatically; calls to this function are therefore not strictly necessary.

RELATED FUNCTIONS

```
sm_save_data();
sm_sv_data(first_field, last_field)
```

svscreen

register a list of screens on the save list

SYNOPSIS

```
int sm_svscreen(screen_list, count)
char *screen_list[];
int count;
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. The number of screens to be added is given by `count`. You may add screens to the list anywhere within your code, however the screen is not actually placed in memory until it is closed for the first time. This means that the time saving factor only comes into play in subsequent openings of the screen. Any data entered into a screen is not saved until the screen is closed.

Screens are removed from the list with `sm_unsvscreen`. You can check to see if a screen is on the save list with `sm_issv`. Checking the list prior to calling `sm_svscreen`, however, is not crucial as any attempt to add a screen that is already on the list has no effect.

This routine saves processing time at the expense of memory. It is best suited for use with screens that both require large amounts of data to be read in from elsewhere (databases, other files, etc.) and do not allow the user to enter data. For instance, if you have a help screen that needs to be populated by a data base and is going to be called up more then once, you can re-display the screen much more quickly by saving the screen in memory.

RETURNS

0 if no error occurred.
1 if registration failed (out of memory).

RELATED FUNCTIONS

```
sm_issv(screen_name);
sm_unsvscreen(screen_list, count);
```

EXAMPLE

```
/*  sm_issv */
/*  sm_svscreen */
/*  sm_unsvscreen */
char *screens[] =
{
```

```
    "start.jam",
    "demo.jam",
    "help.jam"
};

int num_screens = sizeof(screens) / sizeof(char *);

void
save_screens()
{
    /* Put 'screens' onto the save list. */
    sm_svscreen(screens, num_screens);
}

void
release_screens()
{
    /* Remove 'screens' from the save list. */
    return(sm_unsvscreen(screens, num_screens));
}

void
release_screen(name)
char *name;
{
    char *temp[1];
    if (sm_issv(name))
    {
        temp[0] = name;
        sm_unsvscreen(temp,1);
    }
}
```

t_scroll

test whether an array can scroll

SYNOPSIS

```
int sm_t_scroll(field_number)
int field_number;
```

DESCRIPTION

This function returns 1 if the array in question is scrollable, and 0 if not. The argument `field_number` may be any field within the array.

RETURNS

1 if the array is scrolling.
0 if it is not scrolling or if no such `field_number`.

RELATED FUNCTIONS

```
sm_t_shift(field_number);
```

t_shift

test whether field can shift

SYNOPSIS

```
int sm_t_shift(field_number)
int field_number;
```

DESCRIPTION

This function returns 1 if the field in question is shiftable, and 0 if not or if there is no such field.

RETURNS

1 if field is shifting.
0 if not shifting or `field_number` is invalid.

RELATED FUNCTIONS

```
sm_t_scroll(field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Turn on shifting indicators if the screen
 * contains any fields. */

int f;

for (f = sm_inquire(SC_NFLDS); f>0; f--)
{
    if (sm_t_shift (f))
    {
        sm_ind_set (IND_SHIFT);
        sm_rescreen ();
        break;
    }
}
```

tab

move the cursor to the next unprotected field

SYNOPSIS

```
void sm_tab();
```

DESCRIPTION

If the cursor is in a field with a next–field edit and one of the fields specified by the edit is unprotected from tabbing, the cursor is moved to the first enterable position of that field. Otherwise, the cursor is advanced to the first enterable position of the next tab unprotected field on the screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `sm_input`.

RELATED FUNCTIONS

```
sm_backtab();
sm_home();
sm_last();
sm_nl();
```

EXAMPLE

```
#include "smkeys.h"

/* This moves the cursor to the next field. */
sm_tab ();
```

tst_all_mdts

find first modified occurrence

SYNOPSIS

```
int sm_tst_all_mdts(occurrence)
int *occurrence;
```

DESCRIPTION

This function tests the MDT bits of all on–screen and off–screen occurrences of all fields on the current screen, and returns the base field and occurrence numbers of the first occurrence with its MDT set, if there is one. The MDT bit indicates that an occurrence has been modified, either from the keyboard or by the application program, since the screen was displayed (or since its MDT was last cleared by `sm_bitop`).

This function returns zero if no occurrences have been modified. If one has been modified, it returns the base field number, and stores the occurrence number in the variable *addressed* by `occurrence`.

RETURNS

0 if no MDT bit is set anywhere on the screen

The number of the first field on the current screen for which some occurrence has its MDT bit set. In this case, the number of the first occurrence with MDT set is returned in the location referenced by `occurrence`.

RELATED FUNCTIONS

```
sm_bitop(field_number, action, bit);
sm_cl_all_mdts();
```

EXAMPLE

```
#include "smdefs.h"

/* Clear MDT for all fields on the screen. Then write data to the */
   last field, and check that its MDT is the first one set. */

int occurrence;
int numflds;

sm_cl_all_mdts();
numflds = sm_inquire (I_NUMFLDS);
sm_putfield (numflds, "Hello");
if (sm_tst_all_mdts (&occurrence) != sm_inquire(SC_NFLDS)
    sm_err_reset (
    "Something is rotten in the state of Denmark.");
```

udtime

format user–supplied date and time

SYNOPSIS

```
char *sm_udtime(time, format)
struct tm *time;
char *format;
```

DESCRIPTION

This function formats a user–supplied date and/or time according to the specified `format`. `format` is created by using date/time tokens or by using `sm_edit_ptr`. See `sm_sdtime` for details.

This function uses a static buffer which it shares with other date and time formatting functions. The buffer is 256 bytes long. There is no checking for overflow. You should process the returned string, or copy it to a local variable, before making additional function calls.

JAM uses the C header file `time.h` to define `struct tm`.

RETURNS

A pointer to the user date/time in the specified format.
Empty if `format` is invalid.

RELATED FUNCTIONS

```
sm_sdtime(format);
```

EXAMPLE

```
/* Put the date 135 days from now into the field "maturity"  */
#include smdefs.h
time_t tim;
struct tm *matdate;
char *ptr;

/* calculate local time in seconds */
tim = time ((time_t)0) + 135L * 24 * 60 * 60;
matdate = localtime (&tim);
ptr = sm_udtime (matdate, " %0f");
sm_n_putfield ("maturity", ptr);
```

ungetkey

push back a translated key on the input

SYNOPSIS

```
#include "smkeys.h"

int sm_ungetkey(key)
int key;
```

DESCRIPTION

This function saves the translated key given by `key` so that it will be retrieved by the next call to `sm_getkey`. Multiple calls are permitted. The key values are pushed onto a stack (LIFO).

When `sm_getkey` reads a key *from the keyboard*, it flushes the display first, so that the operator sees a fully updated display before typing anything. Such is not the case for keys pushed back by `sm_ungetkey`; since the input is coming from the program, it is responsible for updating the display itself.

RETURNS

The value of its argument, or
–1 if memory for the stack is unavailable.

RELATED FUNCTIONS

```
sm_getkey();
```

EXAMPLE

```
#include "smkeys.h"

/* Force tab to next field */
sm_ungetkey (TAB);
```

unsvscreen

remove screens from the save list

SYNOPSIS

```
void sm_unsvscreen(screen_list, count)
char *screen_list[];
int count;
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. This function is used to remove screens from the save list. The argument `count` specifies the number of screens to be removed from the save list. See `sm_svscreen`.

This function can be used at any point within your code. It is not necessary for the screen to be open at the time of the call. Any memory allocated to hold the screen is freed at the time of the call unless the screen is open. The memory associated with an open screen is de-allocated when that screen is closed. If a screen is not on the save list, a call to `sm_unsvscreen` has no effect.

RELATED FUNCTIONS

```
sm_issv(screen_name);
sm_svscreen(screen_list, count);
```

viewport

modify viewport size and offset

SYNOPSIS

```
void sm_viewport(position_row, position_col, size_row, size_col,
      offset_row, offset_col)
int position_row;
int position_col;
int size_row;
int size_col;
int offset_row;
int offset_col;
```

DESCRIPTION

This function dynamically sizes the current screen's viewport. A viewport has a maximum size of the screen or physical display – whichever is smaller. Use `size_row` and `size_column` to specify the number of rows and columns, respectively.

You can position the viewport anywhere on the physical display. To do this, think of your physical display as a grid made up of rows and columns that are one character apart. The top left corner of your screen monitor is at position row 0, column 0. Now use the arguments `position_row` and `position_col` to specify the coordinates of the viewport.

You may also specify which row and column of the screen should initially appear at top left corner of the viewport. Again Starting at row 0, column 0, count from the top left of the screen to get the coordinates for `offset_row` and `offset_col`.

This function performs range checks on all parameters and suitably modifies them if necessary. In particular, be aware that a non–positive value of `size_row` and `size_col` sets the viewport to the maximum size in that dimension.

EXAMPLE

```
/*      Make current viewport take the full screen      */
void
zoom_screen()
{
    sm_viewport(0,0, -1,-1, 0,0);
}
```

vinit

initialize video translation tables

SYNOPSIS

```
int sm_vinit(video_address)
char *video_address;
```

DESCRIPTION

This routine is called by `sm_initcrt` as part of the initialization process. It can also be called directly by an application program. `video_address` contains the address of a memory resident video file. Such a file must be created by the `vid2bin` and `bin2c` utilities, then compiled into the application.

RETURNS

0 if initialization is successful.
program exit if video file is invalid or if `video_address` is zero and SMVIDEO is undefined.

Note: The variant `sm_n_vinit` has no return value.

VARIANTS

```
sm_n_vinit(video_file);
```

EXAMPLE

```
/* Install a memory-resident video file */

extern char special_vid[];

sm_vinit (special_vid);
```

wcount

obtain number of currently open windows

SYNOPSIS

```
int sm_wcount();
```

DESCRIPTION

This function returns the number of windows currently open. The number is equivalent to the number of windows in the window stack.

To select the screen beneath the current window use:

```
sm_wselect(sm_wcount()-1);
```

This routine is useful when you are bringing another window to the top of the window stack (making the window active) with `sm_wselect`.

RETURNS

The number of windows.
0 if the base form is the only open screen.
–1 if there is no current screen.

RELATED FUNCTIONS

```
sm_wselect(window_number);
```

wdeselect

restore the formerly active window

SYNOPSIS

```
int sm_wdeselect();
```

DESCRIPTION

This function restores a window to its original position in the window stack, after it has been moved to the top by a call to `sm_wselect`. Information necessary to perform this task is saved during each call to `sm_wselect`, but is not stacked. Therefore a call to this routine must follow a call to `sm_wselect` if it is to properly restore the window to its original position. Note that `sm_wdeselect` does not have to be called if the window ordering on the stack is acceptable.

RETURNS

–1 if there is no window to restore.
0 otherwise.

RELATED FUNCTIONS

```
sm_sibling(should_it_be);
sm_wcount();
sm_wselect(window_number);
```

EXAMPLE

```
/* A typical use of the window selection routines is
 * to update information to a window that may (or
 * may not) be covered. For example, suppose that
 * the current time should be maintained on the
 * underlying form. Assume that a field named
 * "curtime" exists on that form. The following
 * code fragments can be used to maintain that
 * field independent of the number of windows
 * currently open above the form.
 */

#include "smdefs.h"

updatetime()
{
    /* quietly select the bottom form */
    sm_wselect (0);
```

```
    /* update system time display */
    sm_n_putfield ("curtime", "");
    /* restore visible window */
    sm_wdeselect ();
    return (0);
}

/* In initialization code: install "updatetime"
   to be called every second. */

    static struct fnc_data afunc = { 0, updatetime,
        0, 10, 0, 0 };
    sm_install (ASYNC_FUNC, &afunc, (int *)0);
```

window

display a window at a given position

SYNOPSIS

```
int sm_r_window(screen_name, start_line, start_column)
char *screen_name;
int start_line;
int start_column;


int sm_r_at_cur(screen_name)
char *screen_name;


int sm_d_window(screen_address, start_line, start_column)
char *screen_address;
int start_line;
int start_column;


int sm_d_at_cur(screen_address);
char *screen_address;


int sm_l_window(lib_desc, screen_name, start_line,
      start_column);
int lib_desc;
char *screen_name;
int start_line;
int start_column;


int sm_l_at_cur(lib_desc, screen_name);
int lib_desc;
char *screen_name;
```

DESCRIPTION

This set of functions is primarily intended to be used by developers who are writing their own executive. To open a window while under the control of the JAM Executive, use a JAM control string or `sm_jwindow`.

Use `sm_d_window`, `sm_l_window`, or `sm_r_window` to display `screen_name` with its upper left-hand corner at the specified line and column. The line and column

are counted *from zero.* If `start_line` is 1, the window is displayed starting at the *second* line of the screen.

Use `sm_d_at_cur`, `sm_l_at_cur`, and `sm_r_at_cur` to display a window at the current cursor position, offset by one line to avoid hiding that line's current display.

Whatever part of the display the new window does not occupy remains visible. However, only the topmost (active) window and its fields are accessible to keyboard entry and library routines. **JAM** does not allow the cursor outside the topmost window. If you wish to shuffle windows use `sm_wselect`.

If the window does not fit on the display at the location you request, **JAM** adjusts its starting position. If the window would hang below the screen and you have placed its upper left–hand corner in the *top* half of the display, the window is simply moved up. If your starting position is in the *bottom* half of the screen, the lower left hand corner of the window is placed there. Similar adjustments are made in the horizontal direction.

When you use `sm_r_window` the named screen is sought first in the memory–resident screen list, and if found there is displayed using `sm_d_window`. It is next sought in all the open libraries, and if found is displayed using `sm_l_window`. Next it is sought on disk in the current directory; then under the path supplied to `sm_initcrt`; then in all the paths in the setup variable `SMPATH`. If any path exceeds 80 characters, it is skipped. If the entire search fails, this function displays an error message and returns.

You may save processing time by using `sm_d_window` and `sm_d_at_cur` to display screens that are memory–resident. Use `bin2c` to convert screens from disk files, which you can modify using `jxform`, to program data structures you can compile into your application. A memory–resident screen is never altered at run–time, and may therefore be made shareable on systems that provide for sharing read–only data. `sm_r_window` and `sm_r_at_cur` can also display memory–resident screens, if they are properly installed using `sm_formlist`. Memory–resident screens are particularly useful in applications that have a limited number of screens, or in environments that have a slow disk (*e.g.* MS–DOS). `screen_address` is the address of the screen in memory.

You may also save processing time by using `sm_l_window` and `sm_l_at_cur` to display screens that are in a library. A library is a single file containing many screens (and/or JPL modules and keysets). You can assemble one from individual screen files using the utility `formlib`. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on the number of paths searched.

The library descriptor, `lib_desc`, is an integer returned by `sm_l_open`, which you must call before trying to read any screens from a library. Note that `sm_r_window` and `sm_r_at_cur` also search any open libraries.

If you want to display a form use `sm_r_form` or one of its variants. Use `sm_close_window` to close the window.

RETURNS

0 if no error occurred during display of the screen;
–1 if the screen file's format is incorrect;
–2 if the screen cannot be found;
–3 if the system ran out of memory but the previous screen was restored;
–5 if, after the screen was cleared, the system ran out of memory.
–6 if the library is corrupted.

RELATED FUNCTIONS

```
sm_close_window();
sm_r_form(screen_name);
sm_jwindow(screen_name);
```

EXAMPLE

```
/* Bring up a window from a library. */

int ld;

if ((ld = sm_l_open ("myforms")) < 0)
    sm_cancel ();
...
sm_l_window (ld, "popup", 5, 22);
...
sm_l_close (ld);
```

winsize

allow end–user to interactively move and resize window

SYNOPSIS

```
int sm_winsize();
```

DESCRIPTION

Calling `sm_winsize` has the same effect as if the end–user had just hit the VWPT (viewport) logical key. The viewport status line appears and the user can move, resize and change the offset of the screen as well as move to any sibling windows. When the end–user hits XMIT (transmit) the previous status line is restored. If you wish to resize the viewport yourself, use `sm_viewport`.

In order for the end–user to able to move from one window to another, the windows must be siblings. Windows are defined as siblings of one another either with `sm_sibling` or by calling up a window as a sibling with a JAM control string. See the sections on "Viewports and Positioning" and "Control Strings" in the *Author's Guide* for further information.

RETURNS

–1 if call fails.
0 otherwise.

RELATED FUNCTIONS

```
sm_sibling(should_it_be);
sm_viewport(position_row, position_col, size_row, size_col,
       offset_row, offset_col);
```

wrecord

write data from a data dictionary record to a structure

SYNOPSIS

```
void sm_wrecord(structure_ptr, record_name, byte_count,
      language)
char *structure_ptr;
char *record_name;
int *byte_count;
int language;
```

DESCRIPTION

This function writes data from fields within the current screen that are part of a common data dictionary record to a C structure. If a field is not on the current screen, then the data is read from the LDB. This routine is commonly used with `sm_rrecord`, which reads data from a structure to a data dictionary record. If you wish to write data only from the current screen, use `sm_wrtstruct`. To write data from a group of consecutively numbered fields, use `sm_wrt_part`. Use `sm_getfield` to write information from an individual field to a string.

To automatically generate a file containing a structure declaration for each data dictionary record, use the `dd2struct` utility. Each structure member is a field within a data dictionary record that is of the type specified in the Screen Editor. See "Data Type" in the *Author's Guide* and `dd2struct` in the *Utilities Guide* for further information.

Once created, the declarations may be treated exactly like any other structure declarations. The argument `struct_ptr` is the address of a variable of one of the structure types generated by `dd2struct`. The argument `record_name` is the name of the data dictionary record, from which the structure was created.

The argument `byte_count` is a pointer to an integer. Upon return from `sm_wrecord`, the value contained in the integer is the number of bytes or characters written to the structure. It is 0 if an error occurred.

The argument `language` stands for the programming language in which the structure is defined. It controls the conversion of string and numeric data.

The following values for `language` are defined in `smumisc.h`:

Language	*Meaning*
S_C_NULL	C with null–terminated strings. This is the most common choice.
S_C_BLNK	C with blank–filled strings. Used for compactness and compatibility with other languages.

RELATED FUNCTIONS

```
sm_putfield(field_number, data);
sm_rrecord(structure_ptr, record_name, byte_count, language);
```

wrotate

rotate the display of sibling windows

SYNOPSIS

```
int sm_wrotate(step)
int step;
```

DESCRIPTION

If two or more sibling windows are on the top of the display, this function may be used to rotate the sequence of the sibling windows.

`step` is a positive or negative integer equalling the number of screen rotations. If `step` is positive, the routine takes the topmost sibling window and makes it the last sibling window for each instance of `step`. If `step` is negative, the routine takes the last sibling window and makes it first. If `step` is zero, no rotations are performed. See the figures below.

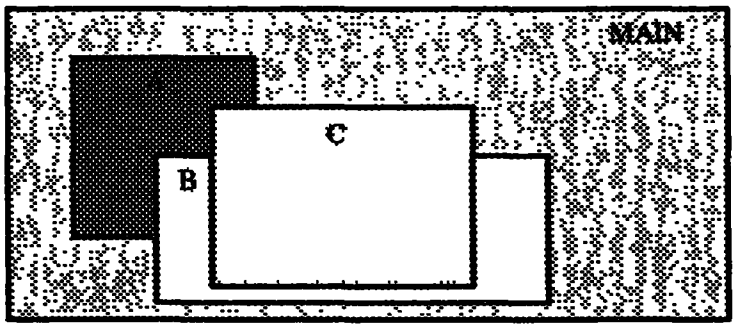


Figure 1: Screens `a`, `b`.and `c` are all siblings.Screen `main` is not a sibling.

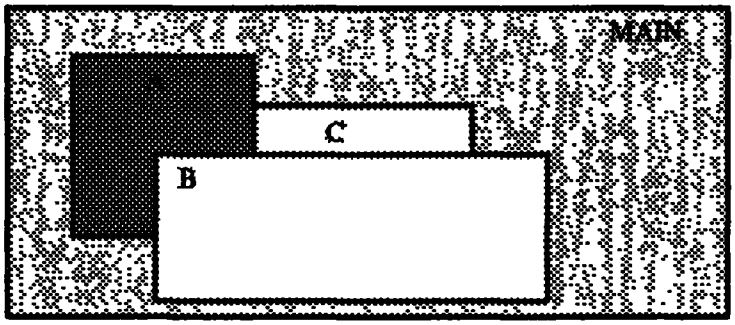


Figure 2: Executing `sm_wrotate (1)` rotates the top sibling to the bottom of the sibling stack. It rotates screen `c` behind the other two sibling windows, leaving screen `b` on top. Screen `main` is not affected.

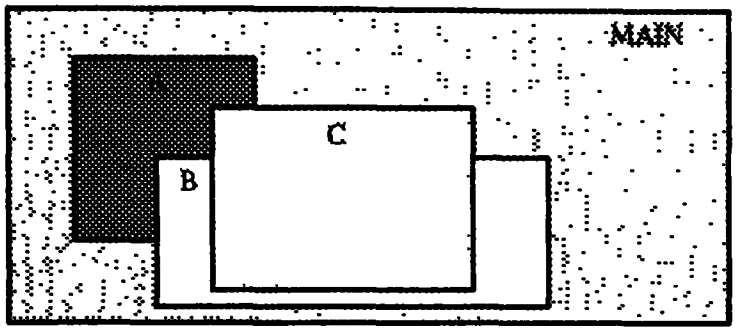


Figure 3: Executing `sm_wrotate (-1)` rotates the last sibling window to the top, putting screen `c` on top. The display is the same as Figure 1.

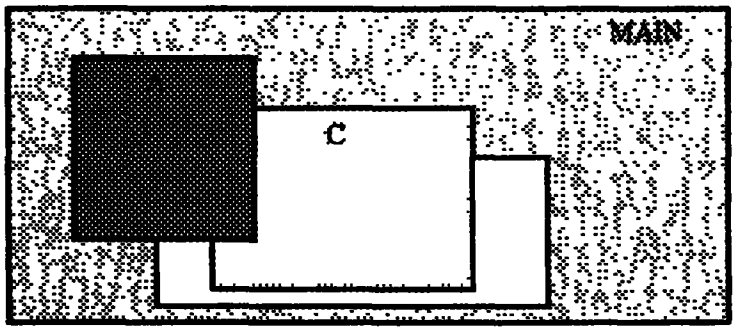


Figure 4: Executing `sm_wrotate (2)` rotates the first two sibling windows off the the top. First it rotates screen `c` to the back, then screen `b`, leaving screen `a` on top.

RETURNS

One less than the number of sibling windows on top of the window stack.
0 if there are no sibling windows

RELATED FUNCTIONS

```
sm_sibling(should_it_be);
```

wrt_part

write part of the screen to a structure

SYNOPSIS

```
void sm_wrt_part(screen_struct, first_field, last_field,
      language)
char *screen_struct;
int first_field;
int last_field;
int language;
```

DESCRIPTION

This function writes the contents of all fields between `first_field` and `last_field` to a data structure in memory. An array and its scrolling occurrences are copied only if the *first* element falls between `first_field` and `last_field`. Group selections are not copied. This routine is commonly used with `sm_rd_part`, which reads part of a structure into the current screen. If you wish to write the contents of all of the fields within the screen use `sm_wrtstruct`. To write information from a data dictionary record, use `sm_wrecord`. Use `sm_getfield` to write information from an individual field to a string.

The structure declaration can be automatically generated from a screen file with the utility `f2struct`. Each member of the structure is a field of the type specified in the Screen Editor. See "Data Type" in the *Author's Guide* and `f2struct` in the *Utilities Guide* for further information.

Once created, the declaration may be treated exactly like any other structure declaration. You can ignore the members that represent fields that do not fall within the bounds of the specified fields. However, the structure definition must contain all of the fields on screen. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`.

The arguments that represent the range of fields to be copied, `first_field` and `last_field` are passed as field numbers.

The argument `language` stands for the programming language in which the structure is defined. It controls the conversion of string and numeric data.

The following values for `language` are defined in `smumisc.h`:

Language	*Meaning*
S_C_NULL	C with null–terminated strings. This is the most common choice.
S_C_BLNK	C with blank–filled strings. Used for compactness and compatibility with other languages.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
sm_putfield(field_number, data);
sm_rd_part(screen_struct, first_field, last_field, language);
sm_wrtstruct(screen_struct, byte_count, language);
```

EXAMPLE

The code example below uses the same screen as the sm_wrt_struct example;

refer to that example for the screen's picture and listing.Here is a header file produced by f2struct from the screen:

```
struct strex
{
    char      date[9];
    char      name[26];
    char      address[3][76];
    char      telephone[14];
};
```

Finally, here is a program that processes the screen using sm_rd_part and sm_wrtpart.

```
#include "smdefs.h"
#include "smkeys.h"
#include "strex.h"

#define C_LANG    0

int main ();
void punt ();

char *program_name;

main (argc, argv)
```

```
char *argv[];
{
    struct strex example;
    int     key;
    char      ebuf[80];

    /* Initialize all structure members to nulls.
     * This is important because we are going to
     * do an sm_rd_part first. */
    example.date[0] = 0;
    example.name[0] = 0;
    example.address[0][0] = 0;
    example.address[1][0] = 0;
    example.address[2][0] = 0;
    example.telephone[0] = 0;

    /* Copy command line arguments, if any, into
     * the structure. */
    switch (argc)
    {
    case 6:
    default:
    /* Ignore extras */
        strcpy (example.telephone, argv[5]);
    case 5:
        strcpy (example.address[2], argv[4]);
    case 4:
        strcpy (example.address[1], argv[3]);
    case 3:
        strcpy (example.address[0], argv[2]);
    case 2:
        strcpy (example.name, argv[1]);
        program_name = argv[0];
        break;
    }
    /* Initialize the screen and copy the structure
     * to it excluding the date field. */
    sm_initcrt ("");
    if (sm_r_form ("strex") < 0)
        punt ("Cannot read form.");
    sm_rd_part (&example, 2, sm_inquire(I_NUMFLDS), C_LANG);
    sm_n_putfield ("date", "");

    /* Open the keyboard to accept new data to the
     * form, and copy it to the structure when done.
     * Break out when user hits EXIT key. */
    sm_d_msg_line ("Enter data; press %KEXIT "
        "to quit.",
        WHITE | HILIGHT);
    do {
        key = sm_input (IN_DATA);
        sm_wrt_part (&example, 2, sm_inquire(I_NUMFLDS),
```

```
            C_LANG);
        sprintf (ebuf, "Acknowledged: byte "
            "count = %d.", count);
        sm_err_reset (ebuf);
    } while (key != EXIT);

    /* Clear the screen and display the final
     * structure contents. */
    sm_resetcrt ();
    printf ("%s\n", example.name);
    for (count = 0; count < 3; ++count)
    {
        if (example.address[count][0])
            printf ("%s\n", example.
             address[count]);
    }
    printf ("%s\n", example.telephone);

    exit (0);
}

void
punt (message)
char *message;
{
    sm_resetcrt ();
    fprintf (stderr, "%s: %s\n", program_name,
        message);
    exit (1);
}
```

wrtstruct

write data from the screen to a structure

SYNOPSIS

```
void sm_wrtstruct(screen_struct, byte_count, language)
char *screen_struct;
int *byte_count;
int language;
```

DESCRIPTION

This function writes the contents of all of the fields within the current screen to a C structure. It does not copy group selections. This routine is commonly used with `sm_rdstruct` which reads data from a structure to all of the fields within the current screen. If you wish to write the contents of a group of consecutively numbered fields into a structure use `sm_wrt_part`. To write information from a data dictionary record, use `sm_wrecord`. Use `sm_getfield` to write the contents of an individual field into a string.

The structure declaration can be automatically generated from a screen file with the utility `f2struct`. Each member of the structure is a field of the type specified in the Screen Editor. See "Data Type" in the *Author's Guide* and `f2struct` in the *Utilities Guide* for further information.

Once created, the declaration may be treated exactly like any other structure declaration. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`. If you specify the type `omit`, data is not written into the field.

The argument `byte_count` is the address of an integer variable. `sm_wrtstruct` stores there the number of bytes copied to the structure.

The argument `language` stands for the programming language in which the structure is defined. It controls the conversion of string and numeric data.

The following values for `language` are defined in `smumisc.h`:

Language	*Meaning*
S_C_NULL	C with null–terminated strings. This is the most common choice.
S_C_BLNK	C with blank–filled strings. Used for compactness and compatibility with other languages.

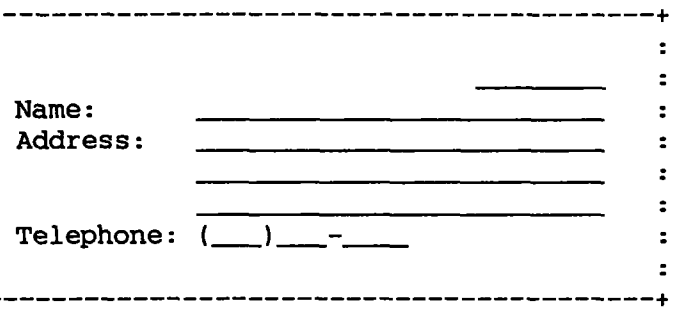
Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
sm_putfield(field_number, data);
sm_rd_struct(screen_struct, byte_count, language);
sm_wrt_part(screen_struct, first_field, last_field, language);
```

EXAMPLE

```
/*
The code example below uses this screen, whose picture and listing
follow.
```




```
            FORM 'strex'
            ------------

FIELD DATA:
-----------

Field number       : 1  (line 2, column 32, length = 8)
Field name         : date
Display attribute  : UNDERLINE HIGHLIGHT WHITE
Field edits        : PROTECTED FROM:  DATA-ENTRY; TABBING-INTO;
                       CLEARING; VALIDATION;
Date field data    : SYSTEM DATE/TIME; 24 HOUR FORMAT = MON2.DATE2.YR2


Field number       : 2  (line 3, column 15, length = 25)
Field name         : name
Display attribute  : UNDERLINE HIGHLIGHT WHITE


Field number       : 3  (line 4, column 15, length = 25)
Field name         : address
Vertical array      : 3 elements; distance between elements = 1
Array field numbers :  3  4  5
Shifting values    : maximum length = 75; increment =  1
Display attribute  : UNDERLINE HIGHLIGHT WHITE
```

```
Field number       : 6  (line 7, column 15, length = 13)
Field name     : telephone
Display attribute  : UNDERLINE HIGHLIGHT WHITE
Character edits    : DIGITS-ONLY
Data type           : CHAR STRING
Initial data:
item 1         : (   )   -

Here is a header file produced by f2struct from the screen above.
*/

struct strex
{
    char      date[9];
    char      name[26];
    char      address[3][76];
    char      telephone[14];
};

/*
Finally, here is a program that processes the screen using sm_rdstruct
and
sm_wrtstruct.
*/

#include "smdefs.h"
#include "smkeys.h"
#include "strex.h"

#define C_LANG   0

int main ();
void punt ();
char *program_name;

main (argc, argv)
char *argv[];
{
    struct strex example;
    int     count,
            key;
    char      ebuf[80];

    /* Initialize all structure members to nulls.
     * This is important because we are going to
     * do an sm_rdstructfirst. */
    example.date[0] = 0;
    example.name[0] = 0;
    example.address[0][0] = 0;
    example.address[1][0] = 0;
```

```
example.address[2][0] = 0;
example.telephone[0] = 0;

/* Copy command line arguments, if any,
 * into the structure. */
switch (argc)
{
case 6:
default:
/* Ignore extras */
    strcpy (example.telephone, argv[5]);
case 5:
    strcpy (example.address[2], argv[4]);
case 4:
    strcpy (example.address[1], argv[3]);
case 3:
    strcpy (example.address[0], argv[2]);
case 2:
    strcpy (example.name, argv[1]);
    program_name = argv[0];
    break;
}

/* Initialize the screen and copy the
 * structure to it. */
sm_initcrt ("");
if (sm_r_form ("strex") < 0)
    punt ("Cannot read form.");
sm_rdstruct (&example, &count, C_LANG);
sm_n_putfield ("date", "");

/* Open the keyboard to accept new data to
 * the form, and copy it to the structure
 * when done. Break out when user hits
 * EXIT key. */
sm_d_msg_line ("Enter data; press %KEXIT\
to quit.", WHITE | HILIGHT);
do {
    key = sm_input (IN_DATA);
    sm_wrtstruct (&example, &count, C_LANG);
    sprintf (ebuf, "Acknowledged: byte "
        "count = %d.", count);
    sm_err_reset (ebuf);
} while (key != EXIT);

/* Clear the screen and display the
 * final structure contents. */
sm_resetcrt ();
printf ("%ld\n", example.date);
printf ("%s\n", example.name);
for (count = 0; count < 3; ++count)
{
```

```
        if (example.address[count][0])
            printf ("%s\n", example.address
                        [count]);
    }
    printf ("%s\n", example.telephone);

    exit (0);
}

void
punt (message)
char *message;
{
    sm_resetcrt ();
    fprintf (stderr, "%s: %s\n", program_name,
        message);
    exit (1);
}
```

wselect

activate a window

SYNOPSIS

```
int sm_wselect(window_number)
int window_number;
```

DESCRIPTION

Although JAM allows you to display multiple windows at one time, only one window may be active. Windows may overlap each other, or may be *tiled* (no overlap). The window at the top of the window stack is the active window, and the only window accessible to library routines and keyboard entry. Use `sm_wselect` to bring a window to the active position on top of the window stack. If any of the referenced window is hidden by an overlying window, it is brought to the forefront of the display. In either case, the cursor is placed within the window. JAM restores the cursor to its position when the screen was most recently de–activated.

The window to be activated is referenced by its number in the window stack. Windows are numbered sequentially, starting from the bottom of the stack. The form underlying all the windows (the base form) is window `0`, the first window displayed is `1` and so forth. Since a screen's number depends on its position on the window stack, calling `sm_wselect` alters a window's number as well as it position on the stack.

Alternatively, windows may be referenced by their screen name with the variant `sm_n_wselect`. If you use this routine, you do not have to worry about keeping track of the non–active window's position on the stack. However, `sm_n_wselect` does not find windows displayed with `sm_d_window` or related functions, because they do not record the screen name.

`sm_wselect` selects sibling windows as a group. If any one of a set of sibling windows is activated by this function, then all of the siblings are brought to the top of the window stack. The selected window will be the active window at the top of this set. Otherwise, the sequence within the set of siblings does not change.

Here are two possible uses for window selection. One use is to select a hidden screen, update it (using `sm_putfield`) and then deselect it (using `sm_wdeselect`). The portion of the hidden screen that is visible is updated with the new data. Because of *delayed write* the update will not be done until the next keyboard input is sought. Another use is to select a hidden screen and open the keyboard. In this case, the selected screen becomes visible, and may hide part or all of the screen that was previously active. In this way you can implement multi–page forms, or switch among several win-

dows that tile the screen (do not overlap). If you want the user to be able to select among screens, define them as siblings.

RETURNS

The number of the window that was made active (either the number passed, or the maximum if that was out of range).
–1 if the window was not found or the window was not open.

VARIANTS

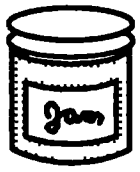
```
sm_n_wselect(window_name);
```

RELATED FUNCTIONS

```
sm_sibling(should_it_be);
sm_wcount();
sm_wdeselect();
```

EXAMPLE

```
/* A typical use of the window selection routines is
 * to update information to a window that may (or
 * may not) be covered. For example, suppose that
 * the current time should be maintained on the
 * underlying form. Assume that a field named
 * "curtime" exists on that form. The following
 * code fragments can be used to maintain that
 * field independent of the number of windows
 * currently open above the form.
 */

#include "smdefs.h"

updatetime()
{
    /* quietly select the bottom form */
    sm_wselect (0);
    /* update system time display */
    sm_n_putfield ("curtime", "");
    /* restore visible window */
    sm_wdeselect ();
    return (0);
}

/* In initialization code: install "updatetime"
   to be called every second. */

    static struct fnc_data afunc = { 0, updatetime,
        0, 10, 0, 0 };
    sm_install (ASYNC_FUNC, &afunc, (int *)0);
```



Chapter 14
Library Function Index

This chapter lists all JAM library functions, sorted by name. Function names appear on the left, and the section of the Function Reference Chapter in which the function is described appears on the right.

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Addendum

for Updates to JAM Release 5.03 Volume 2

for Stratus

Part Number R331–01A

August 3, 1992

Note of Explanation

This addendum describes new features in release 5.03 of **JAM**. This addendum is for Volume 2 of the documentation set. There is a separate addendum for Volume 1.

Descriptions of the features are broken into sections based on the parts of the manual that they affect. In addition, several insertion pages (or A–pages) are included for new library routines in **JAM** 5.03. These pages should be inserted into your **JAM** *Programmer's Guide* at the appropriate location. For example, page A–195 should be inserted before page 195.

Note that the page numbers refer to the August 1, 1991 printing of the **JAM** manual. If you are working with an older manual, insert these pages as appropriate, keeping in mind that the library routines are in alphabetical order.

JPL Guide

Page 13: Return Value from JPL Procedure After an Error

If an error occurs during execution of a JPL procedure (for instance, a math error), the procedure aborts and returns –1. This behavior was no previously documented.

Page 90: Prototyped Functions Called from JPL

Normally, hexadecimal, octal and binary numbers cannot be used in JPL. But when a JPL procedure calls a prototyped function that takes an integer argument, a string to integer conversion takes place. This conversion permits the use of hexadecimal, octal, or binary values as arguments to prototyped functions.

Programmer's Guide

Page 170: New Behavior and Return Codes for `sm_ascroll`

The library routine `sm_ascroll` takes as arguments a field number and an occurrence. It scrolls an array such that the requested occurrence is in the specified field. If the requested occurrence cannot be placed in the specified field because it is one of the first or last occurrences in a non–circular array, then `sm_ascroll` scrolls the occurrence onto the screen and returns the occurrence number of the occurrence that is actually in the specified field.

Page 254: Inquiring Help Level via `sm_inquire`

The global variable `I_INHELP` now contains the level of help that the user is in, instead of just a true/false value. There may be up to five levels of help. Use `sm_inquire` to

query the value of this variable. A return of zero indicates that the user is not in help, a return of 1 through 5 indicates which help level the user is in.

Page 285: sm_keyoption

Certain keys can not be translated via the KEY_XLATE argument to sm_keyoption. These are: INS, REFR, SFTS, LP, and ABORT. They may, however, be disabled via the KEY_ROUTING argument, or intercepted via a keychange function

Page 339: Percent Escapes in sm_query_msg

Percent escapes are now supported for controlling the attributes of query messages. The sequences are the same as those for sm_emsg, and detailed on page 214. Note that %Mu and %Md are not supported. Query messages from JPL can also now use percent escapes.

Page 391: MDT bits and Scrolling Arrays

When lines are inserted or deleted from scrolling arrays via INSL or DELL, the MDT bits for all occurrences after the insertion or deletion are no longer set. In a database application, this prevents the need for unnecessary processing to write potentially large amounts data that have not changed. For large arrays, it can save a significant amount of processing time.

copyarray

copy the contents of one array to another

SYNOPSIS

```
int sm_copyarray(destination_fld, source_fld)
int destination_fld;
int source_fld;
```

DESCRIPTION

This routine copies the contents of the array containing `source_fld` into the array containing `destination_fld`. `source_fld` and `destination_fld` are field numbers. They may be the field number of any of element in the respective array.

The developer is responsible for insuring that the arrays are compatible. Data in source array occurrences that are too long for the destination array are truncated without warning. Data in source array occurrences that are shorter than the destination array's field length are blank filled (with respect for justification).

If the source array has more occurrences than the destination array, the data in the extra occurrences are discarded. If the source array has fewer occurrences than the destination array, trailing occurrences in the destination array are cleared of data (but not de–allocated).

`copyarray` sets the MDT bit and clears the VALIDED bit for each destination array occurrence, indicating that the occurrence has been modified and requires validation.

The variant, `sm_n_copyarray`, searches the LDB for either array if the named field is not found on the screen. However, if the destination LDB item has a scope of 1, meaning that it is a constant, then it is not altered and the function returns –1.

RETURNS

–1 if either field is not found or if the destination array in the LDB has a scope of 1.
0 otherwise.

VARIANTS

```
sm_n_copyarray(destination_name, source_name);
```

RELATED FUNCTIONS

```
sm_clear_array(field_number);
sm_getfield(buffer, field_number);
sm_putfield(field_number, data);
```

key_integer

get the integer value of a logical key mnemonic

SYNOPSIS

```
#include "smkeys.h"

int sm_key_integer (key)
char *key;
```

DESCRIPTION

This function returns the integer value of a **JAM** logical key mnemonic. The value is obtained from the file `smkeys.h`. This function is useful in cases where a function requires the integer value of a key, but cannot access the include files, as in a prototyped function called from JPL. The following table lists the logical key mnemonics:

Logical Key Mnemonics							
EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL	BACK
HOME	DELE	INS	LP	FERA	CLR	SPGU	SPGD
LSHF	RSHF	LARR	RARR	DARR	UARR	REFR	EMOH
INSL	DELL	ZOOM	SFTS	MTGL	VWPT	MOUS	
PF1-PF24		SPF1-SPF24		APP1-APP24		SFT1-SFT24	

RETURNS

the integer value of the logical key mnemonic.
0 if the mnemonic is not found.

RELATED FUNCTIONS

`sm_keylabel(key);`

EXAMPLE

The following example is from JPL. It sets the newline key to act as the tab key. The functions `sm_key_integer` and `sm_keyoption` must be prototyped in order to be called from a JPL procedure.

```
vars ret x y
retvar ret

call sm_key_integer "NL"
cat x ret

call sm_key_integer "TAB"
cat y ret

call sm_keyoption :x 2 :y
return
```

ldb_hash

use hash index for the LDB

SYNOPSIS

```
void sm_ldb_hash();
```

DESCRIPTION

This routine specifies that a hash table should be used to search the local data block. You must call `ldb_hash` before **JAM** initialization, in particular, before you call `sm_ldb_init` to initialize the Local Data Block.

Use of a hash table slightly improves the performance of routines which access the LDB, at the expense of the memory required for the table. This performance improvement includes the LDB merge which is performed the first time a screen with named fields is displayed. The degree of improved performance depends upon the distribution of the names in the LDB, and is greater for LDBs with more entries.

RELATED FUNCTIONS

```
sm_ldb_init();
```

EXAMPLE

```
#include "smdefs.h"

/* create a local data block with a hash index */

sm_ldb_hash();
sm_ldb_init();
```

next_sync

find next synchronized array

SYNOPSIS

```
int sm_next_sync(field_number)
int field_number;
```

DESCRIPTION

Given a field number, this function finds the next array synchronized with the given field, and returns the field number of the corresponding element in that array. The next synchronized array is defined as the one to the right. If `field_number` is in the rightmost synchronized array, the function returns the corresponding element in the leftmost synchronized array (ie– it wraps around the screen).

RETURNS

The field number of the corresponding element in the next synchronized array if there is one.
Otherwise, the field number the function was passed.

set_injpl

allow C routines to access JPL variables & subroutines

SYNOPSIS

```
int sm_set_injpl(mode)
int mode;
```

DESCRIPTION

`sm_set_injpl` allows **JAM** internal routines to access JPL variables, including module and procedure locals, as if they were fields. This is most useful for the **JAM/DB***i* statements `dbi_sql` and `dbi_dbms`, as well as the **JAM** library routine `sm_calc`.

However, `sm_putfield`, `sm_getfield`, or any other user function is not affected by the function call, and there is no means for user code to access JPL variables.

Normally a C routine that is called from JPL, via the JPL `call` statement, does not have access to either the "automatic" variables of the caller (the JPL `proc`) or the "static" variables in the module of the caller. If this routine is called with a `mode` that is non–zero, the C function will have access to both JPL "automatic" and "static" variables. It will also have access to any `proc`'s in the current (the caller's) JPL module. Thus it is as if the C function is embedded bodily within the JPL procedure.

The `mode` remains in effect until the calling JPL procedure is returned to, or `sm_set_injpl` is called again with a `mode` of zero. This means that all subroutines of the C routine will also have access to the current JPL module's variables and procedures. Of course, if the C routine calls a JPL `proc` (e.g. via `sm_jplcall`), the new JPL `proc` *will not* have access to variables in the JPL `proc` that called the C routine.

NOTE: This function should be used with care. For example, since `sm_jwindow` is a C subroutine, it too will have access to the current module's JPL variables and `procs`. In addition any screen entry, exit or validation functions will also have access to these variables and procedures. This can cause some unintended consequences when, for example, a JPL routine opens a screen, and the new screen's entry function calls a JPL `proc`. The JPL processor will look first in the original screen's JPL module (the current module) for the procedure, before it looks in the new screen's JPL module. If it finds a procedure of the same name in the current module it will execute that procedure instead of the procedure in the new screen's JPL. The safest way to use this routine is to set `mode` to a non–zero value when you require access, but then reset it immediately thereafter.

RETURNS

The previous value of `mode`.

JAM Release 5.04 Upgrade Guide

TABLE OF CONTENTS

This addendum describes several new features available in JAM 5.04:

- Menu bars are now available for character–mode applications, including JAM's authoring tools. You can use menu bars developed for graphics environments in character-mode applications, and vice-versa.
- Drop shadows and graying out underlying windows emphasize the active window's appearance.
- The setup variable `SCR_KEY_OPT` is now available for general usage. You can use this variable to enable or disable "remote" scrolling of scrolling arrays.

This addendum contains five chapters:

1 – *Menu Bars* shows how to define menu bars and attach them to your application. It also shows how to manage menu bars at runtime.

2 – *Menu Bar Reference* shows how JAM defines menu bar data and describes the menu bar routines that you can use in your applications.

3 – *Menu Bar Utilities* describes utilities that JAM provides to create and install menu bars.

4 – *Window Display Emphasis* shows how to give drop shadows to windows and gray out their contents.

5 – *Remote Scrolling* shows how to enable scrolling for arrays when the cursor is located outside an arrayed field.

1 Menu Bars

A menu bar is a horizontal menu at the top of the screen that has one or more items. Each item can invoke a pulldown menu—that is, a vertical menu that appears directly below its parent item. Pulldown menu items can themselves invoke submenus. You can nest pulldown menus and their submenus multiple levels deep.

You define menu bars and their pulldowns in ASCII scripts. The script describes the content of the menu bar, the action associated with each item on the menu bar, and its initial status. When you finish defining the menu, you convert the menu script to binary format through the utility `menu2bin`.

After you define a menu bar, you attach it to your application through either JAM's screen editor or JAM library routines. When you attach a menu, you specify whether it is available to the entire program, to other menus, or only to a specific screen or context.

JAM also provides library routines that let you change the content and selection of menu bars at runtime. These are described in Chapter 2 of this addendum.

1.1 MENU BAR SYSTEM

JAM's menu bar subsystem manages the display and behavior of menu bars and their pulldowns. In graphical environments such as Windows and Motif, menu bars use the environments windowing system. In character-mode applications, JAM uses its own resources to display the menus.

If you have menu bars enabled, JAM initializes the menu bar subsystem at startup. It then reads, or loads, menu bars as they are called by the application, either through their associated screens, or through explicit calls to `sm_mn_r_menu` or `sm_mn_d_menu`.

When JAM loads a menu bar, it examines its scope value to decide whether to display it, and when. A menu bar gets its scope value when you attach it to the application. For example, this statement:

```
sm_r_menu (warning, KS_OVERRIDE);
```

specifies to load the menu `warning` at a scope of KS_OVERRIDE.

JAM menu bars can have one of these five scopes:

- KS_FORM associates a menu bar with a screen. This menu bar is displayed with the screen, and with sibling and child windows that lack their own menu bars. JAM can

load only one screen-level menu bar at a time. Thus, if two screens with their own menu bars open in succession, JAM unloads the first screen's menu bar before it loads the second.

You can attach a menu bar to a screen through the screen's keyset field, found on the Screen Characteristics screen. When JAM displays the screen, it automatically loads its menu bar at KS_FORM scope. Alternatively, you can load a screen-level menu bar through a call to `sm_r_menu` or `sm_d_menu` in the screen's entry procedure.

- KS_APPLIC associates a menu bar with the application. Application menu bars are accessible to the application and to screens that lack their own menu bar. You load an application-level menu bar through a call to `sm_r_menu` or `sm_d_menu` in the application's main routine (`jmain.c` or `jxmain.c`), in the area reserved for code to execute before the first screen appears.
- KS_OVERRIDE specifies a menu bar that is independent of any stage of program execution. An override menu has precedence over any other menu bars that are loaded at the same time.

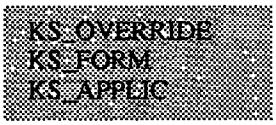
  You can load and unload override menus at any stage of program execution. While multiple override menus can be loaded simultaneously, only one override menu can be active at a time. The active override menu is the most recently loaded one.

  JAM reads all override menus into a save stack, where the program can access them in last-in/first-out order. The save stack can hold up to 10 override menus. JAM sometimes uses the save stack for its own override menus. Consequently, JAM might temporarily push its own override menus on top of user-defined menus loaded earlier.
- KS_MEMRES specifies a menu bar script that is memory-resident and available to other menus at runtime. JAM can maintain as many of these scripts as your system's resources allow. You typically load these scripts at startup, in the application's main routine.

  A memory-resident menu bar is available to other menu bars only as an external menu—that is, a menu that is defined outside the menu that references it. You should install at this scope any menu that is used repetitively by more than one menu.
- KS_SYSTEM specifies the menu bar that JAM uses in its authoring environment. The system menu is loaded by default at startup in the application's main routine, `jmain.c` or `jxmain.c`. Users can then toggle between display of the system menu and the application-level or screen-level menu through the SFTS key or the **Switch Scope** menu item. When JAM switches to the system menu, it closes any screen-level or application-level menu bars that might be loaded, and vice-versa.

Although multiple menus can be loaded simultaneously, only one menu can be displayed at a time. One exception applies: Motif allows simultaneous display of the application-level and screen-level menus.

Within each of the following groups, JAM can simultaneously load menus of these types:



For example, an application can have one or more override menu bars loaded along with one system-level menu. Or, the application can have one screen-level and one application-level menu bar loaded along with several override menu bars. Note that menu bars of KS_SYSTEM scope are loaded to the exclusion of KS_FORM and KS_APPLIC menu bars, and vice-versa.

The previous groups of menu scopes also show their order of precedence. JAM uses this order to determine which of the loaded menus to display. Thus, if an override menu and a screen menu are loaded, JAM displays the override menu. When the override menu closes, JAM displays the screen menu.

If a window without a screen-level menu bar opens, the previously active menu bar remains displayed. This can be the screen-level menu bar from the previous screen, or the application-level menu bar if no screen-level menu bar is loaded.

Motif and OPEN LOOK treat menu bars of different scopes as follows:

- Menu bars can appear on individual screens or on the base screen, depending on their scope and the value of the `formMenus` resource. If `formMenus` is true, the application-level menu bar appears on the base screen while the screen-level menu bar appears local to the screen. Therefore, both can be active at the same time. If a screen without a screen-level menu bar opens, then no menu bar appears local to the screen. Screen-level and override-level menu bars can appear either local to the screen or on the base screen.
- Application-level and system-level menu bars are restricted to the base screen. However, users can always access them as pop up menus by pressing the third mouse button.

1.2
MENU SCRIPTS

When you create a menu bar, you first define it in an ASCII script. You then convert the script to binary with the `menu2bin` utility. A menu script specifies a menu bar and its

pulldowns. The first menu that you specify in a script defines the the top level menu, or menu bar; subsequent menu definitions define pulldown menus and their submenus. You can can nest multiple menus any number of levels deep.

You define a menu with this syntax:

```
menuname[ separator [type]] [ display-option]...
{
   "label" action[ display-option]...
   ...
}
```

Each menu definition begins with a unique identifier, *menuname*. The contents of the menu consist of menu items and separators, enclosed by curly braces `{}`. Except for title items, you can specify items and separators in any order.

Alternatively, a menu script can reference an external menu—that is, a menu that is defined outside the current script. JAM searches for an external menu among currently open menus, then among those menus loaded at the scope `KS_MEMRES`. You specify an external menu as follows:

menuname `external`

The `external` keyword lets you build menu bars in a modular fashion and helps ensure consistency across different menu bars. For example, you can write a script for a pulldown that is repeatedly used by different menu bars. The menu bar scripts can then reference the pulldown as an external menu.

Menu scripts can also include lines of commented text. Prefix each comment line with a pound sign `#`.

The menu bar compiler ignores all white space characters—spaces, tabs, and line returns—except when they separate key words. You can use white space to improve the menu definition's legibility.

The following sections describe individual components of a menu definition.

1.2.1
Menu Name

A menu definition begins with a unique identifier. Each identifier can take one or more display option arguments which JAM applies uniformly to the entire menu. For more information about display options, see "Text Options" on page 9, and "Separator Types" on page 10.

1.2.2
Label

Labels specify the displayed text of menu items. Each label is enclosed in double quote marks (" "). The menu bar compiler accepts labels with up to 255 characters. The label can include backslash escape characters—for example, \n and \t—to specify newlines, tabs, and quotes. JAM uses these formats only if the environment supports them; otherwise, their actual display is terminal-dependent.

To specify a keyboard mnemonic for a menu item, place an ampersand (&) in front of the desired character. Users can select the menu item from the keyboard by typing this character. JAM sets off the mnemonic character according to the display emphasis style that you choose—for example, by underlining or highlighting it. For example, given the following pulldown menu definition:

```
FormMenu
{
    "&New"      key PF1
    "&Open"     control "^jm_filebox file /usr/home * File"
    "&Close"    key PF3   inactive
    "&Save"     key PF3
    "Save &As"  key PF4
}
```

JAM might display the menu like this:

For more information on setting off the display of mnemonic characters in character-mode applications, see "Setting Menu Display and Behavior" on page 16.

1.2.3
Action

The action that you assign to a menu item specifies its behavior. You can use one of the following keywords:

`control` *control-string* — Associates the JAM control string *control-string* with this menu item.

`edit` — Specifies that this menu item invokes the edit pulldown. The edit pulldown typically contains these items: `Cut`, `Copy`, `Paste`, `Delete`, `Select All`. Character-mode applications ignore this action.

`key` *keystroke* — Specifies to return *keystroke* when users select this item. Selection of this menu item is equivalent to pressing the key. The value of *keystroke* can be a JAM logical key. You can also specify a hex, binary or octal number through one of these leading characters:

0x hex
0b binary
0 octal

`menu` *menuname* — Specifies that this menu item invokes the submenu *menuname*.

`separator` [*type*] — Inserts a separator of the specified type between menu items. If you omit the type, JAM draws a single-line separator. See "Separator Types" later in this chapter for information on different separator display options.

`title` — Specifies that *label* is the title of this menu. The title must be the first entry in the menu. Pi/Windows applications ignore the `title` keyword.

`windows` — Specifies that this menu item invokes the Windows menu. This menu lists the names of the open screens. When you select a screen from this menu, JAM brings it to the top of the display. If the selected screen is a sibling of the screen at the top of the window stack, it becomes the top JAM window.

In Windows, the Win-dows menu also contains these items: `Cascade`, `Tile` and `Arrange Icons`. These let you arrange screens and icons within the frame.

In Motif and OPEN LOOK, the Windows menu contains a `raise all` option that raises all JAM screens to the top of the display, and layers them according to the window stack.

In character-based applications, the Windows menu lists only sibling windows, with the last-opened window listed first.

1.2.4

Text Options

You can tell JAM how you want menu items to appear. The following display options are available:

`grayed/greyed` — Mutes, or grays out, the menu item text and prevents users from selecting it.

`help` — Makes a menu item the rightmost item on a menu bar. You can specify this display option for only one menu item, and only if it appears on a menu bar—that is, the script's first, or main, menu definition. If this item is not the last-specified item in the menu definition, JAM rearranges the menu item order so that it appears last.

`inactive` — Inactivates the menu item. When users click on this item, nothing happens.

`indicator` — Indents all menu items to the right and reserves the indented space for the indicator symbol.

`indicator_on` — Turns on the indicator symbol for this item. The indicator—typically an *X* or check mark √—indicates the state of a menu item that serves as a toggle switch.

To change the mark character for character mode applications, assign a new value to `MARKCHAR` in the video file.

Use this option only if you have also specified the display option `indicator`.

`showkey` — If the menu item's action is `key`, shows the keytop label from the key file to the right of the item's text. If the key file has no keytop, then the key mnemonic is shown.

Some options are valid only for certain actions. The following table shows which display options are valid for each action:

Action	Display Options					
	`grayed`	`help`	`inactive`	`indicator`	`indicator_on`	`showkey`
`control`	•	•	•	•	•	
`edit`	•	•	•			
`key`	•	•	•	•	•	•
`menu`	•	•	•			
`title`	•					
`windows`	•	•	•			

1.2.5

Separator Types

JAM offers several ways to separate menu items. An unqualified `separator` action inserts a single or blank line, depending on your system, between the previous and next menu items. You can specify one of the following separator types:

`double` — Inserts a double line.

`double_dashed` — Inserts a double-dashed line.

`etchedin` — In Motif, draws a single line that appears to be etched into the menu. In character-based applications, draws a dotted line.

`etchedout`	In Motif, draws a single line that appears to protrude from the menu. In character-based applications, draws underscores.
`menubreak`	Starts a new line in a horizontal menu, or a new column in a vertical menu.
`noline`	Inserts extra space between the menu items.
`single`	Draws a single line.
`single_dashed`	Draws a single dashed line.

In character-based applications, `single` and `double` can use characters defined in the video file. If no definition exists in the video file, JAM uses its own default values: _, =, | and ||.

You must enter separator types in lower case.

Some separator options are valid only within certain environments. If you specify a separator that your environment does not recognize, JAM uses a blank line (`noline`). The following table shows which separator types are valid for each environment.

Separator type	**Environment**			
	Character	Motif	OPEN LOOK	Windows
`double`	•	•		
`double_dashed`	•	•		
`etchedin`	•	•		
`etchedout`	•	•		
`menubreak`				•
`noline`	•	•		
`single`	•	•	•	•
`single_dashed`	•	•		

1.2.6
Global Menu Settings

Set the separator type and other display options for the entire menu by specifying them after the menu name. For example, if you specify global options `noline` and `showkey`, all separators in the menu default to `noline` and all keys in the menu have `showkey`.

If you specify a global separator type, you can override it for individual separators in the menu.

1.2.7
Comments

You can insert one or more comment lines anywhere in the script. Each comment line must begin with the pound sign `#`.

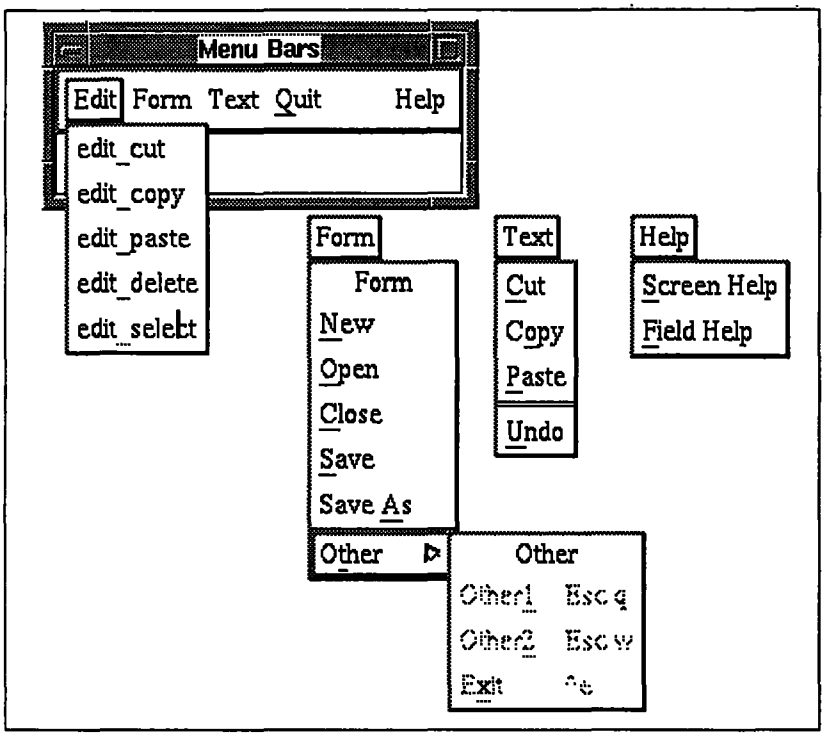
1.3
SAMPLE MENU SCRIPT

This section contains a sample menu script. The figure after it shows how this menu appears in Motif.

```
# The first menu definition specifies the menu bar items

Main
{
        "Edit"  edit
        "Form"  menu FormMenu
        "Text"  menu TextMenu
        "Help"  menu HelpMenu help
        "&Quit" key  0x103
}

FormMenu
{
        "Form"      title
        "&New"      key PF1
        "&Open"     control "^jm_filebox file /usr/home * File"
        "&Close"    key PF3   inactive
        "&Save"     key PF3
        "Save &As"  key PF4
```

```
        ""          separator etchedin
        "O&ther"    menu      OtherMenu
}

OtherMenu grayed showkey
{
        "Other" title
        "Other&1" key PF1
        "Other&2" key PF2
        "E&xit" KEY EXIT
}

TextMenu
{
        "&Cut"    KEY PF1
        "C&opy"   key PF2
        "&Paste"  Key PF3
        ""        sEpArAtOR double menubreak
        "&Undo"   Key SPF1
}

# An external menu is one that is defined elsewhere, either
# in an open menu or at the scope KS_MEMRES.

HelpMenu external
```

This script produces the following output in Motif:



Menu bar and pulldown menus produced by the sample menu script.

1.4
CONVERTING AND STORING MENU BARS

After you write a menu script, use the `menu2bin` utility to convert the script file to binary format. You can store the binary file to a disk file or to a library. This done, you can attach the converted file to an application, as described later.

You can also store menu bar data in memory. To do this, convert the binary file to a C structure with the `bin2c` utility, then register it to JAM with `sm_formlist`. When you store a menu bar's data in memory, JAM compiles it with the application. You can then read these files with `sm_d_menu`.

For information on `menu2bin`, see "Menu Bar Utilities" later in this addendum. For information on `bin2c` and registration of C structures, see the JAM *Programmer's Guide*.

1.5 ATTACHING MENU BARS TO AN APPLICATION

An application that uses menu bars must call each menu bar at the appropriate execution stage. Through JAM's screen editor, you can attach a menu bar to a screen by simply entering the menu name in the Screen Attributes **keyset** field. At runtime, JAM loads the menu bar when the screen opens either as a form or as a window, or when it becomes the topmost window. JAM unloads the menu bar when the screen closes or is superseded by a window with its own menu bar.

Alternatively, you can explicitly load and unload menu bars through calls to one of these routines:

`sm_r_menu`	Reads menu bar data from memory, a library or disk.
`sm_d_menu`	Reads menu bar data from memory. Call this routine only for menu bar scripts that are compiled with the application, as described on page 14, "Converting and Storing Menu Bars."
`sm_c_menu`	Unloads menu bar data and frees the memory associated with it. If the menu bar is still displayed, JAM removes it at the next screen write.

Both `sm_r_menu` and `sm_d_menu` require you to supply a scope value. Menu bars that you attach to a screen through JAM's screen editor automatically get a scope of KS_FORM.

If you attach a menu bar to a screen through JAM routines, you load and unload it in the screen's entry and exit functions, respectively. Override menus (KS_OVERRIDE) can be read at any stage of program execution. Menu bars of all other scopes—KS_APPLIC, KS_SYSTEM, and KS_MEMRES—are typically read in the program's main routine.

1.6
MANAGING MENU BARS

JAM provides library routines to manipulate menu bars and their pulldowns at runtime. For example, you can use sm_mnchange to gray out or activate items according to changes in the screen's context. These library routines are summarized in the following table. See detailed descriptions of each routine in Chapter 2, "Menu Bar Reference."

Runtime routine	Description
sm_mnadd*	Adds an item to the end of a menu.
sm_mnchange*	Alters a menu item (for example, grays out an item).
sm_mndelete	Deletes a menu item.
sm_mnget*	Gets menu item information.
sm_mninsert*	Inserts a new menu item.
sm_mnitems	Gets the number of items on a menu.
sm_mnnew	Creates a menu by name.

* Cannot be prototyped because the routine uses an external data structure.

Because changes to a shared menu bar are passed on to all other screens that use it afterward, be sure that menu bar changes for one screen are correct for all later screens. If you want to omit a menu bar for a specific screen, create a dummy menu bar for it.

You can also refresh the menu to its original state by closing it with sm_c_menu, then reopening it with sm_d_menu or sm_r_menu.

You must prototype any menu bar functions that you call directly from control strings and JPL procedures. The *Programmer's Guide* shows how to prototype and install functions. See also *JPL Guide* for more information on how JPL uses prototyped functions.

1.7
SETTING MENU DISPLAY AND BEHAVIOR

You can control the way menu bars appear and behave in character-based applications by setting various options. You can set these options through sm_option or by editing SMSETUP or SMVARS.

You can specify different display attributes for menu items and their keyboard mnemonic characters so that users can easily distinguish between available and unavailable items. Unavailable items typically appear to be grayed out.

JAM uses the following algorithm for graying menu items and keyboard mnemonic characters:

1. The original display attributes are AND'd with the set of display attributes that you specify to retain.
2. The remaining attributes are OR'd with the attributes that you specify for graying.
3. The result of steps 1 and 2 is exclusive-OR'd with a mask of switch attributes.

You can set different display attributes for a menu item string and its keyboard mnemonic character. If you specify display attributes only for the menu item string, the mnemonic character inherits its attributes.

The following tables show the variables that control display attributes and their default values.

Table 1. Graying attributes.

Variable	Description	Default Attributes
FE_KEEPATTRS	Display attributes that are kept for grayed items.	All attributes
FE_SETATTRS	Display attributes that are set for grayed items.	None
FE_SWATTRS	Display attributes that are toggled when graying is turned on and off.	HILIGHT

Table 2. Keyboard mnemonic character emphasis attributes.

Variable	Description	Default Attributes
AC_KEEPATTRS	Display attributes that are kept for a keyboard mnemonic character	All Attributes
AC_SETATTRS	Display attributes that are turned on for a keyboard mnemonic character.	None
AC_SWATTRS	Display attributes that are switched for a keyboard mnemonic character.	HILIGHT|WHITE

The following table describes options for controlling user interaction and setting menu styles:

Table 3. Settings for menu styles and user interaction.

Variable	Description	Default Attributes
MB_BORDSTYLE	Border style to use in menus (0-9). NOBORDER specifies no border.	1
MB_BORDATT	Menu border attributes.	B_WHITE|BLACK
MB_DISPATT	Display text attributes.	B_WHITE|BLACK
MB_FLDATT	Menu field attributes	B_WHITE|BLACK
MB_HBUTDIST	Distance between two buttons in a horizontal windows-style menu.	2
MB_LINES_PROT	Number of top lines reserved for a menu bar.	1
MB_SYSTEM	Determines whether the System item (==) appears on the menu bar or not. Two options are available: OK_SYSTEM specifies to display System. NO_SYSTEM specifies to omit System.	OK_SYSTEM

2.8
ENABLING MENU BAR SUPPORT

An application that uses menu bars must be enabled to handle them:

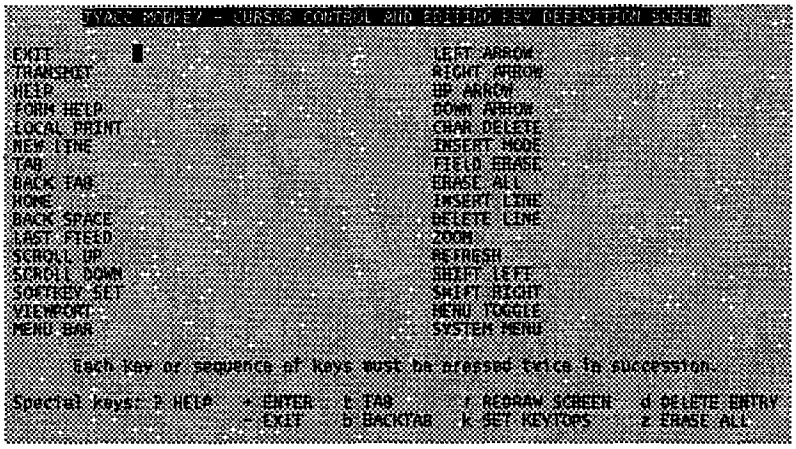
- Create a key file or modify an existing one that defines the keys used to access menu items.
- Optionally, edit your video file and change the default settings for the menu marker character MARKCHAR and submenu indicator SUBMNSTRING.
- Recompile `jmain.c` and, optionally, `jxmain.c`, and rebuild the executable.

2.8.1
Modifying the Key File

You can give users keyboard access to menu items by defining two new types of keys in your key file:

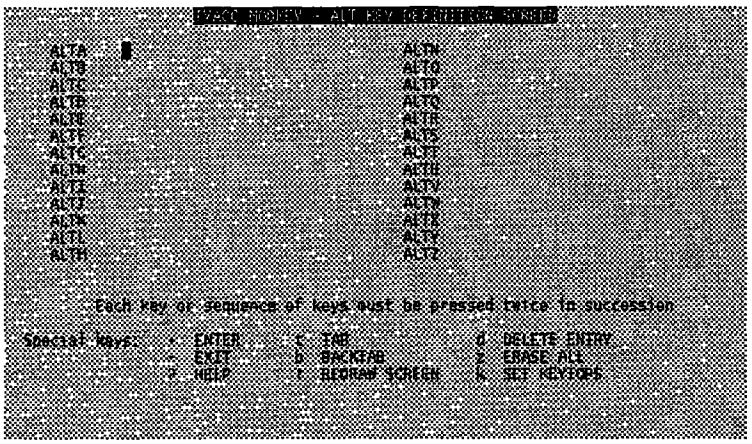
- MNBR lets users access the menu bar. If you are enabling system menu bars, you can also define ALSYS. This gives users keyboard access to the System menu, represented by two equals signs ==.
- ALT keys give users direct access to menu bar items from the screen.

You perform both tasks by modifying an existing key file. You can do this directly, or use JAM's MODKEY utility. Start MODKEY and edit the desired key file. Select *Define Cursor Control and Editing Keys*. JAM displays this screen:



The bottom of the screen contains two entries, MENU BAR and SYSTEM MENU. Define the desired key sequences for MENU BAR and, optionally, SYSTEM MENU—for example, ALT-X and ALT-=. The key sequence for SYSTEM MENU is optional because users can access it indirectly through the MENU BAR key.

After you define the key sequences, exit the screen and return to the menu. Select *Define Alt Keys*. MODKEY displays this screen:



You can now define key sequences that give direct access to menu bar items. For example, you might specify ALT-F to access the menu item **File**.

After you define ALT keys, save the definitions to a file. Then convert this file to binary format with `key2bin`. If the key file is new, add it to the `SMVARS` file.

2.8.2

Modifying the Video File

You can edit the video file as follows:

- Change the value of `MARKCHAR`, which specifies the character used to check menu items. For example, under MS-DOS and JTERM, the following statement specifies the square root symbol (√) as the mark character:

  ```
  MARKCHAR = 0xFB
  ```

- Change the value of `SUBMNSTRING`, which defines the string that menus use to indicate that a menu item invokes a submenu.

After you edit the video file, convert it to binary format with `vid2bin`.

2.8.3

Rebuilding the Executable

After you modify your key and video files, you can build an executable that uses menu bars:

1. Recompile jmain.c with MENUS defined to 1. If you want JAM's authoring environment to include menu bars, recompile `jxform.c` with the same change. Edit the section that specifies optional subsystems to include this statement:

   ```
   #define MENUS 1
   ```

   This tells JAM to initialize the menu bar subsystem at startup.

   Alternatively, add a definition on the compile command line. On UNIX systems, be sure that CFLAGS is set to this value:

   ```
   -DMENUS=1
   ```

   For example:

   ```
   CFLAGS = -I$(HPATH) -O -Aa -DSM_SCCSID -D$(MACHINE_NAME)\
   -DMENUS=1
   ```

2. Relink with the new 5.04 libraries.

1.9
TESTING MENU BARS

You can test menu bars in application mode of JAM if the following conditions are true:

- The SFTS key is defined in the key translation file. This key lets you toggle between the menu bars in your application and JAM's own menu bar.
- If you are testing character-mode applications, you must edit `jxmain.c` to enable menu bars, and rebuild `jxform`. Section 1.8.3 shows how to do this.

1.10
USING MENU BARS AND PULLDOWNS

You can use either the mouse or the keyboard to access menu bars and their pulldowns.

Use the mouse with menus as follows:

1. Click on the menu bar. If you click on a menu item, its pulldown menu appears. If the menu item has no pulldown menu, JAM executes the action associated with the menu item. If you click outside a menu item, JAM selects the menu bar's first item, but does not open its pulldown menu.
2. Exit the menu bar by clicking on the previously active screen.

Access the menu bar from the keyboard through one of keys listed below. Recall that you must first define these keys in the key file.

- MNBR selects the menu bar's first item and leaves its pulldown menu closed.
- ALSYS selects System (=) and opens its menu.
- ALT-*char* selects the menu bar item with the keyboard mnemonic *char*.

You can exit the menu bar by pressing either MNBR or EXIT.

When you switch between screens and menus, JAM maintains the following controls over your work space:

- When you activate a menu bar, JAM saves the state of the screen and keeps all submenus already open.
- When you exit a menu bar and return to its screen, JAM closes all of the menu bar's pulldowns and submenus. Only the menu bar remains visible.
- When the menu bar is active, you cannot manipulate any windows—for example, move or resize them.
- If a message requires user action, JAM prevents you from switching to a menu until you perform the required action.

1.11
MENU BARS AND SOFT KEYS

Soft keys and menu bars are mutually exclusive, because they share the same programmatic hooks. You must choose one or the other. The selection of soft keys versus menu bars is made in the `main` routine, either `jmain.c` or `jxmain.c`, by initializing either soft key support or menu bar support.

JAM provides the `kset2mnu` utility to help you convert keysets to menu bars. This utility converts the keyset to an ASCII menu script. Because the organization of menu bars and keysets can differ greatly, you will probably want to edit `kset2mnu`'s initial output. You can then convert the script to binary format and install it as described earlier.

The `kset2mnu` utility is described in "Menu Bar Utilities" later in this addendum.

2 Menu Bar Reference

This section shows the data structure that JAM uses to modfy or examine menu bar data. It also describes the routines that you can use to create, install, change, and display menu bars. These descriptions appear in alphabetical order.

2.1 MENU BAR DATA

JAM's `item_data` structure lets you change the display or behavior of menu bar items, or to examine a menu item's current state. Some of the runtime routines described in this chapter use `item_data` as a parameter. This structure has the following definition in `smmenu.h`:

```
struct item_data
{
    short *type
    char *label
    short accel
    short key
    char *submenu
    short option
}
```

Each of the structure's members is described below.

`short *type`

Specifies this menu item's type through one of the following defines:

Constant	Value	Description
MT_SEPARATOR	0	Inserts a separator of the specified type between menu items.
MT_TITLE	1	Uses the item's label as the menu title.
MT_KEY	2	Specifies to return a keystroke when users select this item.
MT_SUBMENU	3	Invokes a submenu.
MT_EDIT	4	Specifies that this menu item invokes the Edit pulldown menu.

Constant	Value	Description
MT_WINDOWS	5	Specifies that this menu item invokes the Windows pull-down menu.
MT_CTRLSTRNG	6	Associates a JAM control string with this menu item.

`char *label`

The text of this menu item, ignored if `type` has a value of `MT_SEPARATOR`. Text beyond 255 characters is truncated. The default value is `0`.

`short accel`

The offset of the character in `label` that is used as to select this menu item from the keyboard. The default value is `-1`.

`short key`

The logical key number of the key that is returned on selection of this menu item, valid only if `type` has a value of `MT_KEY`. See `smkeys.h` for a listing of valid key mnemonics. The default value is `0`.

`char *submenu`

If `type` is `MT_SUBMENU`, specifies the menu invoked from this menu item. If `type` is `MT_CTRLSTRNG`, specifies the control string to execute when this item is selected.

`short option`

Display options for this menu item. If the menu item displays the text of `label`—that is, `type` has any value except `MT_SEPARATOR`—you can bitwise OR together the following text display options:

Constant	Value	Description
MO_INDICATOR_ON	0x0200	Turns on the indicator symbol for this item. The indicator—typically a check mark √—indicates the state of a menu item that serves as a toggle switch.
MO_INDICATOR	0x0800	Indents all menu items to the right and reserves the indented space for the indicator symbol.
MO_GRAYED	0x1000	Grays out the menu item text and prevents users from selecting this item.
MO_INACTIVE	0x2000	Makes the entry inactive. When users click on this item, nothing happens.

Constant	Value	Description
MO_SHOWKEY	0x4000	Shows the keytop label from the key file to the right of the item's text. If the key file has no keytop, then the key mnemonic is shown
MO_HELP	0x8000	Makes a menu item the rightmost item on the menu bar. Valid only for one item, and only if it appears on the menu bar.

If the menu item has `type` set to `MT_SEPARATOR`, you can set one of the following options:

Constant	Value	Description
MO_SINGLE	0x0000	Draws a single line.
MO_DOUBLE	0x0001	Inserts a double line.
MO_NOLINE	0x0002	Inserts extra space between the menu items.
MO_SINGLE_DASHED	0x0003	Inserts a single–dashed line.
MO_DOUBLE_DASHED	0x0004	Inserts a double–dashed line.
MO_ETCHEDIN	0x0005	Draws a single line that appears to be etched into the menu.
MO_ETCHEDOUT	0x0006	Draws a single line that appears to protrude from the menu.
MO_MENUBREAK	0x0400	Starts a new line in a horizontal menu, or a new column in a vertical menu.

2.2
MENU BAR ROUTINES

The following routines create, alter, install and display menu bars:

`sm_c_menu`	Closes a menu bar.
`sm_d_menu`	Displays a menu bar stored in memory.
`sm_mncrinit`	Initializes menu bar support.
`sm_mn_forms`	Installs menu bars in memory.
`sm_mnadd`	Adds an item to the end of a menu.
`sm_mnchange`	Changes an item.
`sm_mndelete`	Deletes an item.
`sm_mnget`	Gets information about a menu item.
`sm_mninsert`	Inserts a new item in a menu.
`sm_mnitems`	Gets the number of items in a menu.
`sm_mnnew`	Creates a menu bar.
`sm_r_menu`	Reads and displays a menu bar from memory, a library or disk.

Like other JAM library routines, menu bar routines require the header file `smdefs.h`. Some routines also require you to include other header files, as indicated in the descriptions that follow.

You must prototype any menu bar functions that you call directly from control strings and JPL procedures. JAM's *Programmer's Guide* shows how to prototype and install functions.

The following sections describe menu bar routines in greater detail. The routines are listed alphabetically.

c_menu

close a menu bar

SYNOPSIS

```
#include "smsoftk.h"

int sm_c_menu(int scope);
```

PARAMETERS

`int scope`

Specifies when this menu is available to the application with one of these arguments:

```
KS_FORM
KS_APPLIC
KS_OVERRIDE
KS_MEMRES
KS_SYSTEM
```

DESCRIPTION

This routine closes the menu bar at the specified scope level and frees all memory allocated for it. If the menu bar is displayed, JAM removes it at the next delayed write.

When a menu bar with a scope of `KS_OVERRIDE` closes, JAM pops the next menu, if any, off the override stack.

If the closed menu's scope is `KS_MEMRES`, JAM closes the last menu bar loaded at that scope.

To refresh a menu bar, close it with `c_menu`, then reload it with `r_menu` or `d_menu`.

RETURNS

0 Success.
–2 Menu bar does not exist at this scope.
–3 Menu bars are unsupported or `scope` is out of range.

RELATED FUNCTIONS

`sm_d_menu, sm_r_menu`

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"

/* Close the current JAM window's menu. */

sm_c_menu( KS_FORM );
```

d_menu

load a menu bar that is stored in memory

SYNOPSIS

```
#include "smsoftk.h"

int sm_d_menu(char *menu, int scope);
```

PARAMETERS

`char *menu`

The address of a menu bar stored in memory.

`int scope`

Specifies when this menu is available to the application with one of these arguments:

```
KS_FORM
KS_APPLIC
KS_OVERRIDE
KS_MEMRES
KS_SYSTEM
```

DESCRIPTION

This function can load any menu bar that exists as a C data structure. Use the `bin2c` utility to create program data structures from disk–based menus. You can then compile these into your application and add them to the memory–resident screen list, as described in Chapter 9 of the JAM *Programmer's Guide.*

If a menu bar is already active at the specified scope, JAM compares its name to the value of *menu* and takes one of the following actions:

- If the menu names are the same, the routine returns immediately. Note that you can use this function to refresh the current menu bar display only if you first close it by calling `c_menu`.
- If the menu names are different and `scope` is KS_OVERRIDE, JAM pushes the currently active menu bar into the override stack and makes the newly read menu bar the current menu bar.
- If the menu names are different and `scope` is KS_MEMRES, JAM loads the menu bar along with other memory–resident menus that are loaded and available for use as external menus.

- If the menu names are different and `scope` is KS_SYSTEM, KS_APPLIC, or KS_FORM, JAM closes the previously loaded menu bar and frees the memory allocated for it, then loads the newly read menu bar. If the previous menu bar is displayed, JAM removes it at the next screen refresh.

RETURNS

0 Success.
–1 Not a menu bar.
–3 Menu bars are unsupported or `scope` is out of range.
–5 A malloc error occurred.

If the routine returns with an error, JAM retains the previous menu bar loaded at `scope`, if any.

For all errors except –3, a message is posted to the operator.

RELATED FUNCTIONS

`sm_c_menu, sm_r_menu`

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
  ...
extern char customer_menu[];
  ...

/* Display the customer menu as the application-level menu.
 * Customer_menu was created using bin2c.
 */

sm_d_menu( customer_menu, KS_APPLIC );
```

mncrinit

initialize menu bar support

SYNOPSIS

```
void sm_menuinit();
```

DESCRIPTION

This routine is typically called automatically when you enable menu bars in your application. You enable menu bar support by setting MENUS to 1 in the main routine.

`sm_mmcrinit` sets a global variable to point to a control function. All screen manager functions that need menu bar support check the variable and, if it is non–zero, call indirectly with the request.

You should call this routine explicitly only if you are writing your own executive routine. You call `sm_mncrinit` in the main routine before the call to `sm_initcrt`.

RELATED FUNCTIONS

`sm_mn_forms, sm_mncrinit`

mn_forms

install menu bars in memory

SYNOPSIS

```
void sm_mn_forms();
```

DESCRIPTION

This routine is typically called automatically by JAM's executive. You should call this routine explicitly only if you write your own executive routine and want to load menu bars from memory. You must compile these menu bars into the application and add them to the memory-resident list, as described in Chapter 9 of the JAM *Programmer's Guide*. You can then load these menu bars by calling either `sm_d_menu` or `sm_r_menu`.

You call `sm_mn_forms` in the application's main routine. If you write your own custom executive, you must also call `sm_menuinit` to initialize menu bar support.

RELATED FUNCTIONS

`sm_menuinit, sm_d_menu, sm_r_menu`

mnadd

add an item to the end of a menu bar

SYNOPSIS

```
#include "smsoftk.h"
#include "smkeys.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mnadd(int scope, char *menu_name, struct item_data
*menu_data);
```

PARAMETERS

`int scope`

Specifies when this menu is available to the application with one of these arguments:

`KS_FORM`
`KS_APPLIC`
`KS_OVERRIDE`
`KS_MEMRES`
`KS_SYSTEM`

`char *menu_name`

The name of the menu bar.

`struct item_data *menu_data`

A user-allocated structure that describes the appearance and function of a menu bar item. The description of `item_data` in Section 2.1 shows the values you can assign to this structure, and its default values.

DESCRIPTION

This routine adds an item at the end of the menu specified by `scope` and `menu_name`. The item gets the attributes that you supply to the `item_data` parameter. You assign attributes through identifiers that are defined in `smmenu.h`.

RETURNS

0 Success.
-2 No menu bar exists at this scope.

-3 Menu bars are unsupported or scope is out of range.
-4 menu_name is not found.
-6 Data in item_data is bad.
-7 A malloc error occurred.

RELATED FUNCTIONS

sm_mnchange, sm_mndelete, sm_mnget, sm_mninsert, sm_mnitems, sm_mnnew

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"
#include "smkeys.h"

...

struct item_data *data;
data = ( struct item_data * ) malloc( sizeof( struct item_data )
);

/* Call sm_d_menu w/ a disk resident menu and KS_FORM.
 * Call sm_mnadd to add a title for submenu.
 */

sm_r_menu("mymenu.bin", KS_FORM);
data->type = MT_TITLE;
data->label = "Submenu";
data->accel = -1;
data->key = 0;
data->submenu = 0;
data->option = MO_INDICATOR_ON;
sm_mnadd(KS_FORM, "Submenu0", data);
free(data);

...
```

mnchange

change the text or display attributes of a menu item

SYNOPSIS

```
#include "smsoftk.h"
#include "smkeys.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mnchange(int scope, char *menu_name, int item_no, struct
item_data *menu_data);
```

PARAMETERS

`int scope`

Specifies when this menu is available to the application with one of these arguments:

```
KS_FORM
KS_APPLIC
KS_OVERRIDE
KS_MEMRES
KS_SYSTEM
```

`char *menu_name`

The name of the menu bar.

`int item_no`

A positive integer that specifies the menu item to change, where the first menu item has a value of 0.

`struct item_data *menu_data`

A user-allocated structure that describes the appearance and function of a menu bar item. See "Menu Bar Data" on page 23 for more information on this structure and its values.

DESCRIPTION

Use this function to change a menu item's textual representation or display attributes. JAM modifies the contents of the menu item's data structure through the values that you supply for parameter *data*. For example, you can use this routine to gray out or check an item.

RETURNS

0 Success.
–2 No menu bar exists at this scope.
–3 Menu bars are unsupported or `scope` is out of range.
–4 `menu_name` is not found.
–6 Data in `item_data` is bad.
–7 A malloc error occurred.

RELATED FUNCTIONS

`mnadd, mndelete, mnget, mninsert, mnitems, mnnew`

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"
#include "smkeys.h"

...

/* menu file stored in memory */
extern char mymenu[];

...

struct item_data *data;
data = ( struct item_data * ) malloc( sizeof( struct item_data )
);

/* Call sm_r_menu w/ a disk resident menu and KS_APPLIC.
 * Call sm_mnchange to gray out a menu item in the submenu.
 */
sm_r_menu("mymenu.bin", KS_APPLIC);
data->type = MT_KEY;
data->label = "NewItem";
data->accel = 3;
data->key = PF1;
data->submenu = 0;
data->option = MO_GRAYED|MO_SHOWKEY;
sm_mnchange(KS_APPLIC, "Submenu0", 0, data);
free(data);

...
```

mndelete

delete a menu bar item

SYNOPSIS

```
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mndelete(int scope, char *menu_name, int item_no);
```

PARAMETERS

`int scope`

Specifies when this menu is available to the application with one of these arguments:

KS_FORM
KS_APPLIC
KS_OVERRIDE
KS_MEMRES
KS_SYSTEM

`char *menu_name`

The name of the menu bar.

`int item_no`

A positive integer that specifies the menu item to delete, where the first menu item has a value of 0.

DESCRIPTION

This routine deletes the item specified by `item_no`, `menu_name`, and `scope` from the menu bar. The first item on a menu has an `item_no` value of zero.

RETURNS

0 Success.
–2 No menu bar exists at this scope.
–3 Menu bars are unsupported or `scope` is out of range.
–4 `menu_name` is not found.
–5 `item_no` is not found.

RELATED FUNCTIONS

`sm_mnadd, sm_mnchange, sm_mnget, sm_mninsert, sm_mnitems, sm_mnnew`

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

...

int count;

/*
  Delete the last item from the application menu
  called "customer"
*/

if ((count = mnitems( KS_APPLIC, "customer" )) > 0)
  sm_mndelete( KS_APPLIC, "customer", count );

...
```

mnget

get information about a menu bar item

SYNOPSIS

```
#include "smsoftk.h"
#include "smkeys.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mnget(int scope, char *menu_name, int item_no, struct
item_data *menu_data);
```

PARAMETERS

`int scope`

Specifies when this menu is available to the application with one of these arguments:

`KS_FORM`
`KS_APPLIC`
`KS_OVERRIDE`
`KS_MEMRES`
`KS_SYSTEM`

`char *menu_name`

The name of the menu bar.

`int item_no`

A positive integer that specifies the menu item to get information on, where the first menu item has a value of 0.

`struct item_data *menu_data`

A user–allocated structure that describes the appearance and function of a menu item. See "Menu Bar Data" on page 23 for more information on this structure and its values.

DESCRIPTION

This function fills the fields in the `item_data` structure with the corresponding data of the menu item. Note that you must create buffers for the label and submenu elements of the structure that are large enough to hold the label and submenu names, as in the example shown later. The maximum length is 255 characters.

RETURNS

0 Success.
–2 No menu bar exists at this scope.

–3 Menu bars are unsupported or `scope` is out of range.
–4 `menu_name` is not found.
–5 `item_no` is not found.

RELATED FUNCTIONS

`mnadd, mnchange, mndelete, mninsert, mnitems, mnnew`

EXAMPLE

```
#include "smdefs.h"
#include "smmach.h"
#include "smmenu.h"
#include "smsoftk.h"

...

/* menu file stored in memory */
extern char mymenu[];

...

char buf1[100], buf2[100];

struct item_data *data;

data = (struct item_data *) malloc( sizeof( struct item_data ) );

data->label = buf1;
data->submenu = buf2;

/*  Call r_menu with a disk resident menu.
 *  Call mnget to get an override-level menu bar item.
 */

sm_r_menu("mymenu.bin", KS_OVERRIDE);
sm_mnget(KS_OVERRIDE, "Main", 0, data );
free(data);

...
```

mninsert

insert a new menu item

SYNOPSIS

```
#include "smsoftk.h"
#include "smkeys.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mninsert (int scope, char *menu_name, int item_no, struct
item_data *menu_data);
```

PARAMETERS

`int scope`

Specifies when this menu is available to the application with one of these arguments:

`KS_FORM`
`KS_APPLIC`
`KS_OVERRIDE`
`KS_MEMRES`
`KS_SYSTEM`

`char *menu_name`

The name of the menu bar.

`int item_no`

A positive integer that specifies the menu item to follow the inserted item, where the first menu item has a value of 0.

`struct item_data *menu_data`

A user–allocated structure that describes the appearance and behavior of the menu item to insert. See "Menu Bar Data" on page 23 for more information on this structure and its values.

DESCRIPTION

This routine inserts a new menu bar item before the menu item specified by `item_no`, `menu_name`, and `scope`, using the data in the menu bar structure `item_data`.

RETURNS

0 Success.
–2 No menu bar exists at this scope.

–3 Menu bars are unsupported or scope is out of range.
–4 menu_name is not found.
–6 Data in item_data is bad.
–7 A malloc error occurred.

RELATED FUNCTIONS

mnadd, mnchange, mndelete, mnget, mnitems), mnnew

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"
#include "smkeys.h"

...

struct item_data *data;

data = ( struct item_data * ) malloc( sizeof( struct item_data )
);

/* Call sm_r_menu w/ a disk resident menu and KS_FORM.
 * Call sm_mninsert to insert a submenu.
 */

sm_r_menu("mymenu.bin", KS_FORM);
data->type = MT_SUBMENU;
data->label = "NewItem";
data->accel = 3;
data->key = 0;
data->submenu = "Submenu1";
data->option = MO_INDICATOR;
sm_mninsert(KS_FORM, "Main", 1, data);
free(data);

...
```

mnitems

get the number of items on a menu bar

SYNOPSIS

```
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mnitems(int scope, char *menu_name);
```

PARAMETERS

`int scope`

Specifies when this menu is available to the application with one of these arguments:

```
KS_FORM
KS_APPLIC
KS_OVERRIDE
KS_MEMRES
KS_SYSTEM
```

`char *menu_name`

The name of the menu bar.

DESCRIPTION

This routine returns the number of items on the menu bar specified by `menu_name` and `scope`.

RETURNS

If successful, the function returns the number of items in the menu; otherwise, it returns one of these values:

–2 No menu bar exists at this scope.
–3 Menu bars are unsupported or `scope` is out of range.
–4 `menu_name` is not found.

RELATED FUNCTIONS

`sm_mnadd, sm_mnchange, sm_mndelete, sm_mnget, sm_mninsert, sm_mnnew`

EXAMPLE

```
#include "smdefs.h"
#include "smmach.h"

...

int ret;

/* Call sm_r_menu w/ a disk resident menu and KS_OVERRIDE.
 * Call sm_mnitems to get the number of items on the menu bar, and

 * place the number in the current field.
 */

sm_r_menu("mymenu.bin", KS_OVERRIDE);
ret = mnitems(KS_OVERRIDE, "Main");
   sm_n_itofield( "number", ret );

...
```

mnnew

create a menu

SYNOPSIS

```
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mnnew(int scope, char *menu_name);
```

PARAMETERS

`int scope`

The scope of the menu bar to create. Supply one of these values:

`KS_FORM`
`KS_APPLIC`
`KS_OVERRIDE`
`KS_MEMRES`
`KS_SYSTEM`

`char *menu_name`

The name of the menu bar.

DESCRIPTION

This routine creates a submenu in the menu bar structure at the specified scope level. After you call this routine, specify its contents by calls to `sm_mnadd` and `sm_mninsert`. After you create the menu and its contents, attach it to an existing menu by creating an item that invokes it, through a call to `sm_mnadd` or `sm_mninsert`

RETURNS

0 Success.
–2 No menu bar exists at this scope.
–3 Menu bars are unsupported or `scope` is out of range.
–4 `menu_name` is not found.
–7 A malloc error occurred.

RELATED FUNCTIONS

`sm_mnadd, sm_mnchange, sm_mndelete, sm_mnget, sm_mninsert, sm_mnitems;`

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"
#include "smkeys.h"

...

int ret;
struct item_data *data;

data = ( struct item_data * ) malloc( sizeof( struct item_data )
);

/* Call sm_r_menu w/ a disk resident menu and KS_OVERRIDE.
 * Call sm_mnnew to create a new menu bar .
 * Call sm_mnadd to add items to it and finally add this new menu
 * to the menu displayed as a submenu.
 */

sm_r_menu("main.bin", KS_OVERRIDE);
ret = sm_mnnew(KS_OVERRIDE, "NewItem");
   if ( ret == 0 )
   {
     data->type = MT_TITLE;
     data->label = "Submenu";
     data->accel = -1;
     data->key = 0;
     data->submenu = 0;
     data->option = MO_INDICATOR_ON;

     sm_mnadd(KS_OVERRIDE, "NewItem", data);

     data->type = MT_SUBMENU;
     data->label = "I";
     data->accel = 0;
     data->key = 0;
     data->submenu = "Submenu1";
     data->option = MO_INDICATOR;

    sm_mnadd(KS_OVERRIDE, "NewItem", data);

     data->type = MT_SUBMENU;
     data->label = "NewItem";
     data->accel = 3;
```

```
    data->key = 0;
    data->submenu = "NewItem";
    data->option = MO_INDICATOR;

    sm_mnadd(KS_OVERRIDE, "Main", data);
  }
free(data);
...
```

r_menu

read a menu bar from memory, a library or disk

SYNOPSIS

```
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

int sm_r_menu(char *menu_name, int scope);
```

PARAMETERS

`char *menu_name`

The name of the menu bar to read.

`int scope`

Specifies when this menu is available to the application with one of these arguments:

KS_FORM
KS_APPLIC
KS_OVERRIDE
KS_MEMRES
KS_SYSTEM

DESCRIPTION

When you call this routine, JAM first looks for the specified menu bar in the memory–resident screen list, next in any open libraries, and finally on disk in the directories specified by the argument to `sm_initcrt` and by `SMPATH`.

If a menu bar is already active at the specified scope, JAM compares its name to the value of *menu_name* and takes one of the following actions:

- If the menu names are the same, the routine returns immediately. Note that you can use this function to refresh the current menu bar display only if you first close it by calling `sm_c_menu`.
- If the menu names are different and `scope` is KS_OVERRIDE, JAM pushes the currently active menu bar into the override stack and makes the newly read menu bar the current menu bar.
- If the menu names are different and `scope` is KS_MEMRES, JAM loads the menu bar along with other memory–resident menus that are loaded and available for use as external menus.

- If the menu names are different and `scope` is KS_SYSTEM, KS_APPLIC, or KS_FORM, JAM closes the previously loaded menu bar and frees the memory allocated for it, then loads the newly read menu bar. If the previous menu bar is displayed, JAM removes it at the next screen refresh.

RETURNS

0 Success.
–1 Not a menu bar.
–2 No menu bar exists at this scope.
–3 Menu bars are unsupported or `scope` is out of range.
–4 `menu_name` is not found.
–5 A malloc error occurred,

In the case of an error the previously displayed menu bar remains displayed.

For all errors except –3 a message is posted to the operator.

RELATED FUNCTIONS

`sm_c_menu, sm_d_menu`

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

...

/* Read in the company menu and display it at the form level. */

sm_r_menu( "company.bin", KS_FORM );

...
```

3 Menu Bar Utilities

JAM has two utilities for creating menu bars:

- `menu2bin` converts an ASCII menu script into a binary menu file.
- `kset2mnu` converts a JAM keyset into an ASCII menu script. For detailed instructions on creating menu bar scripts, see "Menu Bars" earlier in this addendum.

The following sections describe these utilities in detail.

menu2bin

convert ASCII menu scripts to binary format

SYNOPSIS

`menu2bin [-pv] [-e` *ext*`]` *menufile...*

OPTIONS

`-p` Places the binary files in the same directories as the input files.

`-v` Lists the name of each input file as it is processed.

`-e` Appends *ext* to the output file name. The default extension is `bin`.

DESCRIPTION

The `menu2bin` utility converts ASCII menu scripts into binary format. Menu scripts are created as text files. Chapter 1, "Menu Bars", shows how to write a menu script.

Menu binary files can be placed in libraries with the `formlib` utility. Refer to the JAM *Utilities Guide* for more information.

ERRORS

`Too many menu definitions. Max is 128.`
Only 128 menu definitions may be included in one menu script.

`Too many item definitions. Max is 128.`
Only 128 item specifications may be included in one menu definition.

`Cannot create '%s'`
`Error writing '%s'`
An output file could not be created, due to lack of permission or perhaps lack of disk space. Correct the file system problem and retry the operation.

`Neither '%s' nor '%s' found.`
An input file was missing or unreadable. Check the spelling, presence and permissions of the file in question.

`Error in '%s' line '%d'` *error-type*
The syntax of your script on the specified line is incorrect. The value of *error-type* specifies one of these errors:

```
Expected left brace '{' after menu name.
No right brace '}' found before EOF.
No menu name specified.
Expected quoted item label.
Missing action.
```

```
Unknown action '%s'.
Unknown option '%s'.
No key specified.
Bad key '%s'.
Bad escape sequence '%s'.
Undefined submenu '%s'.
More than one option of this type (%s).
More than one accelerator character assigned.
Accelerator character at end of string - Ignored.
Menu '%s' is on menu bar so cannot be used as submenu.
```

kset2mnu

convert keysets into ASCII menu scripts.

SYNOPSIS

`kset2mnu [-pv] [-e` *ext*`] keyset...`

OPTIONS

`-p`	Places the binary files in the same directories as the input files.
`-v`	Lists the name of each input file as it is processed.
`-e`	Appends *ext* to the output file name. The default extension is `mnu`.

DESCRIPTION

The `kset2mnu` utility converts keysets into menu scripts and stores it in an ASCII text file. The utility converts a keyset according to these rules:

- The first row in the keyset becomes the menu bar.
- Subsequent rows become submenus. Submenus are named "Row*x*", where *x* is the row number.
- The `SFT`*x* key (goto row *x*) becomes an entry for the submenu named Row*x*.
- The `SFTN` (next row) and `SFTP` (previous row) keys become entries for the submenus named Row*{i+1}* or Row*{i–1}*, where *i* is the current row.

Because menu bars and keysets are often organized according to different principles, the converted menu bar often requires manual editing. For example, keyset items in the first row typically invoke actions, while menu bar items usually invoke pulldowns whose items invoke actions.

When you finish editing the menu bar script, convert it to binary format with `menu2bin` and attach it to the application.

ERRORS

`Soft key '%s' designates a nonexistent submenu.`
The keyset contains a `SFT`*n* key for a row that does not exist. Remove the offending key from the keyset and reconvert it.

`Neither '%s' nor '%s' found.`
An input file was missing or unreadable. Check the spelling, presence, and permissions of the input file.

`Cannot create '%s'`
`Error writing '%s'`
An output file could not be created, due to lack of permission or disk space. Correct the file system problem and retry the operation.

4 Display Emphasis

JAM now has two display options that let you emphasize the current, or active, screen:

- *Drop shadows* appear to cast a shadow from the active screen shade over underlying screens.
- *Graying* changes the display attributes of all screens except the active one according to a predefined algorithm—for example, highlights turn off and colors change to monochrome.

An application can use both methods singly or together.

Drop shadows and graying change only the display attributes of the background screens; the actual contents are unaffected. You can specify which display attributes to preserve and which new attributes to use for the grayed data.

To use display emphasis, set JAM as follows:

- Specify the emphasis style to use—drop shadows or graying.
- Specify the display attributes of a grayed object.

The following sections show how to perform both tasks.

4.1 SPECIFYING EMPHASIS STYLE

You specify which emphasis style to use by setting the value of the configuration variable `EMPHASIS`. You can set this value in the configuration file as follows:

```
EMPHASIS=style
```

You can also reset the emphasis style at runtime through the library function `sm_option`:

```
sm_option(EMPHASIS, style);
```

Note that after `sm_option` changes the emphasis style, you must call `sm_rescreen` to repaint the display.

You can specify one of the following values for `style`:

- `NONE` disables display emphasis.

- GRAYBKGD grays background screens. Only the active screen retains its original display attributes.
- DROPSHADOW draws a shadow at the topmost screen's right and bottom edges. The drop shadow is two columns wide and one line deep. The right shadow starts one space below the screen's top edge, while the bottom shadow starts two columns from the screen's left edge. The bottom shadow is indented two spaces from the left edge of the screen. The shadow is formed by graying the underlying text.

4.2
SETTING GRAY ATTRIBUTES

Whether you use graying or drop shadows, you must set the display attributes that JAM uses for the underlying objects. You set these through two video file variables:

- EMPHASIS_KEEPATT specifies the attributes that a grayed object retains. This variable intially has all attributes enabled except HILIGHT.
- EMPHASIS_SETATT specifies the attributes that the grayed object acquires. This variable intially has two attributes enabled: REVERSE and DIM.

You can reset EMPHASIS_KEEPATT and EMPHASIS_SETATT either in the video file, or through the runtime function sm_pset. For example, the following statement sets graying with attributes DIM and WHITE:

```
sm_pset(V_EMPHASIS_SETATT, "DIM WHITE");
```

After you call sm_pset, call sm_rescreen to update the display.

See section 4.6 in the JAM *Configuration Guide* for display attribute names.

5 Remote Scrolling

You can now configure JAM applications to allow or disallow scrolling data inside an array when the cursor is positioned outside that array. This is particularly useful for character-mode applications in which users need to view off-screen data in arrays that are tab-protected.

You enable or disable remote scrolling by assigning one of these values to the setup variable SCR_KEY_OPT:

- SCR_NEAREST, the default, enables remote scrolling. This causes the nearest scrollable array to scroll when the user presses a scrolling key.
- SCR_CURRENT allows users to scroll array data only when the cursor is in that array. Scrolling keys are inactive when the cursor is outside a scrollable array.

INDEX

M

O

P

R

S

V

W

JAM/DBi

Please forward comments regarding this document to:

Technical Publications Manager
JYACC, Inc.
116 John Street
New York, NY 10038

JAM/DBi

Contents

1 Introduction

1.1 What is JAM/DBi?

JAM/DB*i* provides an easy-to-use, standard, portable interface between JAM applications and a variety of popular SQL-based databases.

JAM is a development tool which provides a prototyping, development and testing environment for the rapid development of software applications.

SQL (standard query language) is a tool which provides end users with a non-procedural, easy-to-use means of accessing databases. SQL assumes little or no programming skills. SQL is also an emerging standard. This means that users can move from one machine, operating system and/or database to another with little or no retraining in database access methods.

JAM/DB*i* ties together the cost-effectiveness of JAM application development with the power of SQL-based databases. And, if the developer so chooses, it can all be done without ever writing a line of third-generation, procedural programming code.

A key feature of JAM/DB*i* is that it uses the query language syntax (SQL) of the database you are using. Instead of learning a new syntax, users continue to use the syntax with which they are already familiar.

JAM/DB*i* is easy to use, because data retrieval and update are accomplished through JAM data dictionary definitions. The user is not required to understand the lower level operation of the database or JAM. When data are retrieved (using an SQL SELECT) they are placed in data dictionary or screen elements whose names correspond to the database table column names. If column names do not match data dictionary elements, they may be "coerced" or mapped using database-supported column mapping (or alias mapping if a database does not support column mapping.) For updates (SQL INSERT or UPDATE), the user simply specifies the names of the data dictionary or screen elements to be inserted as host variables.

There are many advantages to the JAM/DB*i* solution:

- it makes linking application screens to databases trivial
- it eliminates the need to know the low level access routines of a database system
- it virtually guarantees portability of an application across many different hardware and operating system platforms
- it provides a standard means of moving applications from one database to another with no changes to screens and very few (if any) changes to SQL scripts
- it enables a developer to prototype an application with real links to a database without ever writing a line of code. Later the developer can go back to the same application and build in procedural calls to provide additional functionality.

1.2 What Makes Up JAM/DBi?

The media, file names and contents of your JAM/DB*i* files are dependent on the hardware and operating system for which you ordered JAM/DB*i*.

Every version of JAM/DB*i* should have the following files:

- one or more files (usually in binary format) that contain the object code of JAM/DB*i*;
- utilities;
- a makefile for a JAM/DB*i* version of `jxform`;
- a makefile for a JAM/DB*i* version of JAM;
- makefiles for utilities (if necessary).

The actual file names for your machine and operating system are described in Appendix A of this document.

1.3 What Is in This Document?

This document describes

- what JAM/DB*i* is;
- how to use JAM/DB*i* in building applications that use supported databases,

- and how to use JAM/DB*i* in specific hardware and operating system environments.

This document assumes that the reader has a basic knowledge of the target computer system, databases, JAM and `jxform`. Except for details related to building a JAM/DB*i* link, this manual does not provide any details about how to use `jxform` or JAM.

1.4 What Is the JAM/DBi Development Cycle?

Before you can do any development, you have to build a copy of JAM that includes the links to the database interface. This executable is `jamdbi`. Appendix A describes the procedure you should follow. You may also build a version of `jxform` with links to a database. This executable is `jxdbi`. Some versions of `jxdbi` running under PC-DOS will not have enough memory to do development. In such cases, you will have to use `jxform` without DBi linked in and test using `jamdbi`.

There are four steps to building a working application with JAM/DB*i*.

- First, you build a JAM screen using `jxform` or `jxdbi`. This involves defining JAM fields and providing an access path between a JAM application and JAM/DB*i*. In building a JAM screen, you have several ways to gain access to JAM/DB*i*. The quickest way is to make a screen entry call to JPL. The second way is to associate a JAM/DB*i* function with a keyboard key. The third means of access is through attached functions. The details of making these calls are presented below in a demonstration scenario.
- Second, you should add the fields that you defined in your JAM screens to the JAM data dictionary. This ensures that the data brought in from the database to one screen will be available in all other screens, too.
- Third, you need to create a set of JPL statements that are the actual SQL calls to the database. The name of the JPL file must be the same as the one specified in the JAM screen entry specification or in an associated key. We assume that you already have a set of database tables with some data to access.
- Fourth, you test your program.

1.4.1 A Development Scenario

In the next few pages, we will walk you through a complete JAM/DB*i* development cycle. For our scenario we assume you are using the ORACLE RDBMS with the following table:

MYDATA

with the following fields:

NAME ADDRESS CITY STATE.

To create MYDATA, go into SQLPLUS and enter:

```
create table mydata
(name char (30),\
address char (30),\
city char (15),\
state char(2));
```

You can insert several values into this table with the following statements:

```
insert into mydata values
('John Doe',\
'1312 Geary Blvd',\
'San Francisco','CA');
insert into mydata values
('Jane Roe',\
'505 West End Ave',\
'New York','NY');
insert into mydata values
('Edgar Woe',\
'3712 Rio Grande Blvd',\
'Albuquerque','NM');
insert into mydata values
('Amy Snow',\
'1400 Lakeshore Dr',\
'Chicago','IL');
```

Now exit SQLPLUS and go into `jxform` (or `jxdbi`).

Press <Shift PF5> and go into **FORMAKER**.

When prompted for the name of a form, enter MYTEST and press <XMIT>.

Press <PF3> to define a border and background colors. Depending on the characteristics of your terminal, assign any values you want but leave the size of the form as the default size.

Press <XMIT>.

Press <SHIFT PF1> to associate a function key with a set of SQL instructions. A window will appear with the names of the keyboard keys. Move your cursor down to PF1 and at the prompt for the name of an entry function enter:

```
^jpl myjpl.jpl
```

Press <XMIT>.

Move your cursor down two lines and over several columns.

Now draw in an underscore with 30 characters and, when finished, move back one space. Press <XMIT>.

Press <PF4> to define field attributes. Assign the following characteristics:

Type <A> and enter the field name NAME (upper case for ORACLE!).

Press <XMIT>.

Type <S> to change the size of the field to an array.

Tab to number of elements and enter "10" for an array of 10 occurrences.

Tab to horizontal and enter <N> (for "No").

Press <XMIT> to confirm these field attributes.

Type <E> (for "Exit") to return to the form window.

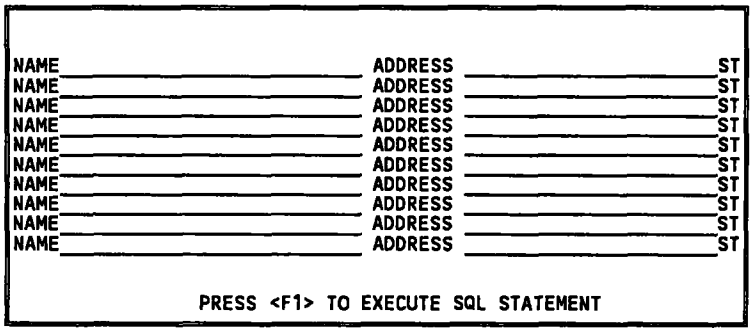
Move the cursor over a few spaces and repeat the same process for a field of 30 characters and name it ADDRESS.

It is important for ORACLE that the field names be entered in upper case. (Some databases do not care, and some want the entry to be in lower case. These considerations are described in Appendix B.)

Move your cursor to the next-to-the-last line on the screen and about 20 columns from the left and enter the following text:

```
PRESS <F1> TO EXECUTE SQL STATEMENT
```

You should have a screen that looks something like this when you press <SHIFT PF3>:



Press <PF1> to return to the form window.

Press <EXIT> and type <S> to save your form. When the "form saved" message appears, type <SPACE> and <E> to exit.

Using any text editor, create an ASCII text file called

```
'myjpl.jpl'
```

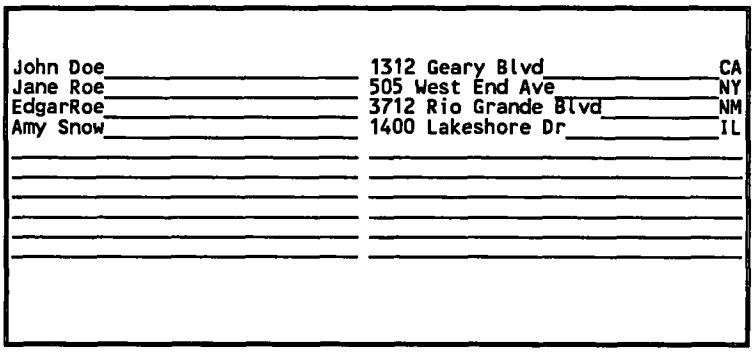
and enter the following lines:

```
DBMS LOGON username password
SQL SELECT NAME,ADDRESS,STATE FROM MYDATA
DBMS LOGOFF
RETURN 0
```

username and *password* are your name and password on your ORACLE RDBMS. (We assume you have been granted resources to create a table.)

Exit your editor and type JAMDBI MYTEST. When the screen comes up, press <PF1> to execute the JPL/SQL statements.

If all went well, your screen should look like this:



To exit, press <EXIT>.

2 Accessing JAM/DBi

There are two basic ways of accessing JAM/DB*i* functions: through JPL statements and direct C language function calls.

The advantage of using JPL is that

- it is easier to prototype an application;
- it is easier to see what is going on in a particular function;
- it is easier to change a function,
- and you do not have to compile anything in order to see your application run.

The advantage of using C calls is that

- the reading and first parsing of JPL statements are eliminated;
- your SQL calls are embedded and cannot be changed (or erased!) by end users,
- and your applications are more secure.

Both of these access methods are described below.

2.1 JPL Calls

There are two types of DB*i* statements in JPL. The JPL DBMS statements begin with the keyword DBMS and JPL SQL statements begin with the keyword SQL.

(Note: Under the ShareBase version of JAM/DB*i* there is an additional JPL verb - IDL - which works much like SQL. The IDL dependent features are described in Appendix B of the JAM/DB*i* ShareBase documentation.)

2.1.1 JPL DBMS Calls

DBMS statements are used for either database specific functions such as logging on, or for controlling the JAM/DB*i* environment such as setting error levels. Examples of JPL DBMS statements are:

```
DBMS LOGON DEMO JIM MYPASSWD
```

```
DBMS COUNT MYCOUNT
DBMS ERROR
```

2.1.2 JPL SQL Calls

SQL statements are used for standard SQL calls such as SELECT, INSERT, UPDATE, COUNT, etc. Examples of the use of JPL SQL statements are:

```
SQL SELECT NAME, ADDRESS, CITY FROM MYDATA
SQL SELECT NAME FOR UPDATE FROM EMP WHERE\
                    EMPLOY_NUM=678
```

2.2 Embedded C Calls

If desired, DBMS and SQL calls may be hidden and passed directly to JAM/DB*i*. However, the developers who do this must be careful to *test the return codes* and handle errors properly.

There are two functions that can be called, namely, **dbi_dbms** and **dbi_sql** (summarized below). The appropriate function call corresponds to whether the argument being passed is a DBMS or a SQL statement.

NAME

dbi_dbms -parse and execute a DBMS statement

SYNOPSIS

```
int dbi_dbms(dbms_statement)
char * dbms_statement;
```

DESCRIPTION

Parses, validates and executes a DBMS statement. The DBMS statement must be syntactically the same as a JPL DBMS statement, *but without the JPL verb DBMS at the beginning.*

RETURNS

0 if no error
-1 if an error.

NAME

dbi_sql -parse and execute a SQL statement

SYNOPSIS

```
int dbi_sql(sql_statement)
char * sql_statement;
```

DESCRIPTION

Parses, validates and executes a SQL statement. The SQL statement must be syntactically the same as a JPL SQL statement, *but without the JPL verb SQL at the beginning*. The statement must also be syntactically correct for the database being used.

NOTE: Because no JPL parsing is done on SQL statements called by this function, there is no colon substitution of variables.

RETURNS

0 if no error
-1 if an error.

---

3 Initialization

Most databases require a logon procedure or a function call. The actual parameters used in the logon depend on the database. See Appendix B for details.

The syntax of the logon command is:

```
DBMS LOGON <other arguments>
```

If, for example, one were using Gupta Technologies' SQLBase and the data dictionary variables DBNAME (for database name), USER (for the user name) and PASSWORD (for the user's password), the logon command would look like this:

```
DBMS LOGON :DBNAME :USER :PASSWORD
```

The phrases :DBNAME, :USER and :PASSWORD are variables which JAM/DB*i* will replace with correct values from the JAM data dictionary. The process of variable substitution is described below.

The obverse of the logon command is *logoff*. There are no arguments to the logoff command:

```
DBMS LOGOFF
```

You should always execute a LOGOFF to disconnect properly. Otherwise, you may create inconsistencies in your database.

4 Fetching and Inserting Data

JAM/DB*i* permits you to move data between a JAM application and a supported database. JAM/DB*i* may insert data into, or update data from, a JAM screen, a JAM data dictionary and/or a supported database.

All data manipulation in JAM/DB*i* is done using the JPL verb SQL. The SQL verb in turn expects to be passed a SQL data manipulation string using column-based instructions such as SELECT, INSERT, UPDATE and DELETE, or table- or view-based instructions such as CREATE, ALTER or DROP.

JAM/DB*i* manipulates all SQL statements dynamically, creating temporary data storage space, doing any requisite data conversions and, in the case of a SELECT, moving data into a JAM screen or the JAM data dictionary.

JAM/DB*i* pre-parses your SQL statement and makes any necessary variable substitutions. JAM/DB*i* then passes on to your database system the SQL statement you asked JAM/DB*i* to prepare.

All database errors are trapped and will be displayed (depending on the environmental controls set by the developer; see Section 5 below). Warnings may be ignored, depending on the environmental handling set by the application developer.

IMPORTANT DEPENDENCY: JAM/DBi SQL syntax is native to the database you are using. For example, some databases convert column names to lower (upper) case and return data with the lower case mapping. Users who define JAM screen or data dictionary names in upper (lower) case with such databases will not find any data being passed back and forth between JAM and the database.

4.1 Variable Substitution

A SQL statement such as

```
SQL SELECT NAME, ADDRESS FROM EMP
```

will always search and return all instances of data with name and address in the table EMP. In many cases, however, such a SQL statement needs to be qualified. For example:

will always search and return all instances of data with name and address in the table EMP. In many cases, however, such a SQL statement needs to be qualified. For example:

```
SQL SELECT NAME, ADDRESS FROM EMP WHERE\
                        NAME='JOHN'
```

In this case, SQL will search for and return all instances of data where name is equal to JOHN.

For most applications, the qualifying value in the WHERE clause of a SQL statement is not known until runtime. To handle such situations, JAM/DB*i* allows dynamic variable substitution in SQL statements.

Substitution variables are variables in a SQL statement that are replaced with values from the JAM application screen or the data dictionary. These correspond to standard SQL host variables. However, JAM/DB*i* provides an extended capability in that substitution variables can contain any character string for substitution. This means that substitution variables may contain whole or partial SQL statements for substitution (see below).

4.1.1 Defining Substitution Variables

A substitution variable is identified by a preceding colon (e.g. :MYVAR). Colons in the middle of a word (e.g. MY:VAR) will be ignored. When a colon is detected, the word following it is used to search the field list on the JAM screen and then the JAM data dictionary. (A double colon (::) can be used to suppress substitution.)

If the substitution variable name is found, the corresponding value is inserted into the SQL statement in place of the substitution variable name.

The substitution variable may be a single element, a complete array or an element of one. If an array name is specified without reference to a specific array element, all non-blank fields in the array are inserted in place of the substitution variable. The inserted fields of a complete array are separated by single spaces.

Examples of using substitution variables follow:

```
SQL INSERT INTO EMP VALUES\
(':NEWNAME',:SALARY,':START_DATE',:EMPLOY_NUM,'HIRE')
```

```
SQL UPDATE EMP SET SALARY=:SALARY_TABLE[4]\
WHERE EMPLOY_NUM=:EMPLOY_NUM
SQL :SQL_STATEMENT
```

The first SQL statement adds a new row of data to the database table EMP with the JAM string variables :NEWNAME and :START_DATE, the JAM numeric variables :SALARY and :EMPLOY_NUM and a fixed string HIRE.

The second statement modifies an existing row or rows of data in the table EMP. It changes all instances of SALARY in EMP to the value of element 4 of a JAM array field called SALARY_TABLE. In that array, the column EMPLOY_NUM is equal to JAM data field EMPLOY_NUM.

The third SQL statement replaces the JAM/DB*i* SQL variable :SQL_STATEMENT with whatever is found in the JAM data field SQL_STATEMENT. In this case, presumably, SQL_STATEMENT would hold an entire SQL statement. Users may also use nested bindings with JPL. See JAM JPL Programmer's Guide for instructions.

Note that SQL statements in JPL are not terminated by a semi-colon (;).

4.2 Column Mapping and Aliases

When a SELECT statement is executed, JAM/DB*i* copies returned values into JAM screen or data dictionary fields, if any. For JAM/DB*i* to do this, there *must* be a one-to-one mapping between SQL column names and JAM field names. For example, a SQL statement retrieving data from EMPLOY_NAME will attempt to place the returned data into a JAM field of exactly the same name. (Names are truncated in JAM to 31 characters.) If such a field is not found, JAM/DB*i* will ignore the returned value for that column.

It is because of this mapping that users can invoke a SQL statement like the following:

```
SQL  SELECT * FROM EMP
```

However, there are times when a one-to-one mapping is not possible or it is too constraining. In such cases, many databases permit a remapping of names. For example, if the database column was named EMP_NAME and the JAM field was named EMPLOYEE_NAME (perhaps because EMP_NAME was already used for some other purpose), the developer can remap the association of database and JAM field names. This is done within the SQL statement itself. Using our example where the database column name is EMP_NAME, the JAM fields EMP_NAME and EMPLOYEE_NAME are defined and the developer does not want to change the current JAM field value of EMP_NAME, the appropriate SQL statement would be:

```
SQL SELECT EMP_NAME EMPLOYEE_NAME
FROM EMP WHERE EMPLOY_NUM=:EMPLOY_NUM
```

Note that there is *no comma* between EMP_NAME and EMPLOYEE_NAME. It is the absence of the comma that permits the parser to map the association of the column name EMP_NAME with the JAM field name of EMPLOYEE_NAME. A comma between the two names would have caused a SQL SELECT error if EMPLOYEE_NAME were not also in the table EMP.

IMPORTANT DEPENDENCY: There are some databases that DO NOT SUPPORT COLUMN REMAPPING. Check in Appendix B to determine whether your database supports column remapping. If your database does not support remapping, JAM/DB*i* provides an alternative means of accomplishing the same thing using a DBMS command called DBMS SELECT_ALIAS.

SELECT_ALIAS is available *only* in those databases that do support remapping. A SELECT_ALIAS must be executed just before the SQL SELECT statement to be parsed. SELECT_ALIAS must be used if *any* of the column names in the SQL SELECT statements do not directly correspond to the JAM field names (e.g., if the DBMS column name is NAME and the JAM field name is CLIENT_NAME). An example of SELECT_ALIAS is:

```
DBMS SELECT_ALIAS CLIENT_NAME, TEST, -, RESULT
```

for the SQL statement:

```
SQL SELECT NAME, GRADE, AGE, SCORE FROM XYZ
```

The hyphen in the SELECT_ALIAS means that no remapping is required for the third column name in the SQL SELECT (i.e., AGE). There is a one-to-one correspondence between the number of arguments in SELECT_ALIAS and the number of columns specified in a SQL SELECT.

4.3 Fetching

Most of the information you need to fetch data with SELECT has been specified above. However, there are several implementation details that are important.

1. When retrieving multiple rows of data, JAM/DB*i* will determine the maximum number of rows that can be retrieved at one time. In the event a JAM field is defined as an array in the screen or the data dictionary, JAM/DB*i* will take the minimum number of defined occurrences of the field in the screen or the data dictionary. While developers are strongly discouraged from creating arrays in a form and the data dictionary of the same name but different size (measured in terms of array element occurrences), JAM/DB*i* will protect the developer by returning the lesser of the occurrences. To continue retrieving data, see the DBMS CONTINUE command below.
2. The maximum number of rows of data that JAM/DB*i* will return in a single fetch is based on the smallest number of array occurrences of any single data element in the fetch. As an example, if you execute SQL SELECT NAME, ADDRESS, STATE FROM EMP, and NAME and ADDRESS have been defined in a JAM application as arrays of 15 occurences each and STATE as an array of 10 occurrences, then the maximum number of rows returned will be 10 at a time.
3. Some SQL developers may be used to forcing the closing of a SQL data storage area (called a SQL cursor). In JAM/DB*i*, you do not have to force the closing of a SQL cursor. JAM/DB*i* will automatically open and close cursors.
4. Because JAM Version 4.0 permits fields to be defined on a screen and not necessarily in a data dictionary, JAM/DB*i* uses the following order of precedence when searching for a field name and its characteristics:

a. Screen variables
b. Data dictionary variables

5. If after searching these lists JAM/DB*i* cannot match a field name with a SQL column name, JAM/DB*i* will ignore that SQL column in the subsequent retrieval of data.
6. JAM fields may be defined anywhere. Subsequently, a SQL statement may call for the retrieval of data where some fields are defined only in the JAM screen, where others are defined only in the JAM data dictionary and yet others are defined in both places.
7. Some users of JAM/DB*i* Version 3.X will find that some JAM/JPL DBMS verbs are no longer supported (DIAG in Informix and AUTOCOMMIT in ORACLE).

4.4 Continuing

If a SELECT returns more rows than can be placed into a JAM field array (wherever it is defined), you can subsequently continue to retrieve more data by attaching a DBMS CONTINUE statement in a JPL file.

Moreover, if a user explicitly calls a CONTINUE after SQL has indicated there are no more rows to fetch, JAM/DB*i* will not access the database. To control this, the developer should set the environment variable ERROR (see Section 5) and constantly check the current value of the ERROR variable for a 'no more rows' condition. The actual value returned is database dependent.

Alternatively, users can use the DBMS COUNT function to check the number of rows returned by the SQL SELECT and maintain their own current count of how many rows are left in the fetch. (This is obviously more problematic for databases that permit forward and backward retrieval of rows.)

4.5 Next/Cancel/Flush

4.5.1 Next

Some databases support multiple commands to be issued in a single string. If your database supports this feature, JAM/DB*i* will attempt to process such strings. If, however, a command (other than the last command in a sequence) returns data (via

SELECT) and you are required to issue a DBMS CONTINUE, the original command will be suspended. To continue processing a multicommand string from a suspended state, issue the following:

```
DBMS NEXT
```

4.5.2 Cancel

If there are multiple commands in a single string and processing has been suspended as described above, the remaining commands may be canceled with the following command:

```
DBMS CANCEL
```

4.5.3 Flushing

Some database systems using multicommand strings (e.g., Britton Lee) require that, when a SELECT has returned multiple rows and not all rows are fetched by JAM/DB*i*, the application must explicitly flush unread rows before another command is processed. Since there are no side effects to flushing, you may issue a flush command even if you are not sure whether there are any more rows left. The syntax for this command is:

```
DBMS FLUSH
```

5 JAM/DBi Environment

JAM/DB*i* provides several functions to control processing, error trapping and database maintenance. These functions frequently use native functions provided by the database system.

5.1 Determining the Number of Rows a SELECT Will Find

Since JAM/DB*i* keeps track of only the rows read but not of how many are held in a cursor (some databases do not easily provide that information), you can find out how many records will be found by a given SQL SELECT by issuing the SELECT COUNT command, as explained below.

First create a variable in the JAM data dictionary to hold the return value. Let's assume we have such a variable, called TOTAL.

```
SQL SELECT COUNT (EMPLOY_NUM) TOTAL\
FROM EMP WHERE EMPLOY_NUM > :EMPLOY_NUM
```

A subsequent SQL statement that uses the same table and WHERE clause will return the same number of rows as the number in TOTAL.

5.2 Determining the Number of Returned (Read) Rows

When a SQL SELECT is issued, there is no guarantee of how many rows of data the database will return. Frequently, it is important to know either that no rows were returned or how many rows must be processed.

Note that the DBMS COUNT function does not return the total number of rows found by SQL for a specific SELECT. (Use a SQL COUNT to do that.) It returns the number of rows read into memory from the current SQL cursor.

To find out how many rows were returned, define either a JAM screen field or data dictionary field to hold the row count (e.g. MYCOUNT), and issue the following command:

```
DBMS COUNT MYCOUNT
```

After each SELECT and any subsequent CONTINUE, JAM/DB*i* will place the returned row count into MYCOUNT.

DBMS COUNT can also be used for SQL DELETE and SQL UPDATE.

5.3 Specifying a Start Row

Some databases do not offer any means to page backwards through the database, nor does the current version of JAM/DB*i* provide any direct support for backward paging (or redirected start and end). To help you around this problem, JAM/DB*i* provides a means for reading a predetermined number of records and discarding them before beginning to read the records that you want to see.

Let's assume that you want to page backwards through the database. Just issue the following commands:

```
VARS RUNNING_COUNT
CAT RUNNING_COUNT "0"
DBMS START :RUNNING_COUNT
DBMS COUNT MYCOUNT
RETURN 0
```

Issue a SQL SELECT COUNT command:

```
SQL SELECT COUNT (EMPLOY_NUM) TOTAL FROM EMP
```

This number can be used to make sure that you do not create a starting row value greater than the number of rows that can be returned.

Then issue your SELECT statement:

```
SQL SELECT EMPLOY_NAME,EMPLOY_NUM FROM EMP
```

After each SELECT or CONTINUE, add the value in MYCOUNT to RUNNING_COUNT.

Let us also assume that you have an array that holds 15 rows of data and that the SQL statement has returned the value of 50 into TOTAL. Furthermore, assume that you have read 3 pages (or 45 rows of data), of which the first 2 pages are lost. To go back one page of data, issue the following JAM/JPL command:

```
MATH RUNNING_COUNT=RUNNING_COUNT - 30
IF RUNNING_COUNT <"0"
{
CAT RUNNING_COUNT "0"
}
DBMS START:RUNNING_COUNT
```

```
SQL SELECT EMPLOY_NAME,EMPLOY_NUM FROM EMP
RETURN 0
```

Of course, if rows are added to or deleted from the table by other users, the starting point will be only proximate to the top of each page as the user pages forwards.

5.4 Error Processing

By default, JAM/DB*i* displays any database errors at the bottom of the screen. If you want to control error processing yourself, use the DBMS ERROR command to define a field to hold the database return code and, optionally, an error message. For example, to save the return value of a database status, create a field called MYERROR and issue the following:

```
DBMS ERROR MYERROR
```

If you want to store a return error message from the database, you can specify a data dictionary element (and optionally an occurrence). For example, if you have an array ERROR_MSG with 20 occurrences and you want to store a database error in the 8th occurrence, use the following statement:

```
DBMS ERROR MYERROR ERROR_MSGS[8]
```

To return to the default condition, just issue the following:

```
DBMS ERROR
```

Users should note that in JAM/DB*i* Version 4.0 SQL errors will not terminate JPL processing. In Version 3.X, on the other hand, if error trapping was turned off, SQL errors would stop JPL processing.

Note: The DBMS ERROR function, if active, passes on only those errors that are returned by the database. Errors in JPL syntax will report an error and terminate the JPL procedure. Moreover, some databases do not return error messages, and in some cases users are required to fetch messages from a database file on the basis of an error number.

5.5 Warning Processing

By default, JAM/DB*i* ignores database warnings. To catch and process warnings, first create a JAM data dictionary variable (for example, MYWARNING) to store the warning message and then issue the command:

```
DBMS WARN MYWARNING
```

You can also use JAM arrays. Note that warning formats vary from database to database. See Appendix B for details. You can return to the default status by issuing the following:

```
DBMS WARN
```

5.6 Begin/Commit/Rollback

If your database supports "before image journaling", you may create sets of transactions that may be written to the database at once. Such sets provide the opportunity to "rollback" an entire set of interrelated transactions if one of the SQL transactions fails. The way this procedure works varies from database to database. The basic process includes marking the beginning of a transaction, executing one or more transactions and then, after testing the return codes of these transactions, executing either a COMMIT or ROLLBACK. In some databases, there is no need to mark the beginning of a transaction since all transactions since the last COMMIT or ROLLBACK will be affected by a COMMIT or ROLLBACK.

The syntax for these commands is:

```
DBMS BEGIN
DBMS COMMIT
DBMS ROLLBACK
```

6 JAM/DBi Utilities

Version 4.0 of JAM/DB*i* provides two utilities to aid in program development. The first utility - `f2tbl` - creates a database table from a binary JAM form. The second utility - `tbl2f` - creates a basic JAM form from a database table definition.

NAME

`f2tbl` - form to table utility

SYNOPSIS

`f2tbl <JAM file name>`

DESCRIPTION

This utility program will take a user-specified JAM form in binary file format, identify JAM fields and data types and create a table in the user's database with those fields.

This version of `f2tbl` maps a one-to-one correspondence between form and table. Future versions will permit a form to map to different tables.

The user is prompted for the name of the table to be created. If the database being used requires a database name, user name and/or password, the user will also be prompted for that information.

`f2tbl` will exclude JAM control fields (i.e., those defined with a `jam_` definition and fields that are not named).

`f2tbl` will create a maximum of 50 columns (JAM fields) in a table.

The only two data types supported are character and integer. All other data types will be converted to one of these types.

NAME

tbl2f - table to form utility

SYNOPSIS

tbl2f

DESCRIPTION

This utility program will take a database table and create a binary JAM form with named fields corresponding to the fields in the database table.

This version of tbl2f maps a one-to-one correspondence between table and form. Future versions will permit multiple tables to map to a single form.

If the database requires a database name, user name and/or password, the user is prompted for such information. tbl2f will prompt the user for a table name. tbl2f will then prompt the user for a form name. The default is the database table truncated (if necessary) to a valid file name length with the extension .JAM.

A maximum of 24 fields will be created. tbl2f will inform the user of how many fields were created or how many were dropped from the specification if the number of columns exceeds 24.

The three data types supported are character, integer and float. All other data types will be converted to one of these types. Note that LONG VARS (variable length) will be created but data will be truncated. Future versions of JAM/DB*i* will support LONG and RAW vars.

Each field will have a maximum on-screen size of 20, but fields larger than 20 characters will be made into shifting fields.

New Features in JAM/DBi for ORACLE

Release 4.8

Contents

I. Multiple Cursors

In the following,

cursor_name is an identifier, consisting of non-blank characters, with maximum same as a **JAM** variable.

sql_statement is an SQL statement, possibly containing variables (syntax is database specific).

var is a **JAM** variable reference. It may have one of the following three forms:

id or *id[int]* or *id[id]*

where id is a **JAM** identifier, and int is an integer. The last 2 forms are array element references. Throughout this document, "**JAM** variable" will be used to denote any of the different kinds of variables supported in **JAM**, i.e., JPL variables, screen fields, and data dictionary variables. When the same name is defined in more than one place, JPL variables have the highest order of precedence, and data dictionary definitions the lowest.

The **JAM**/DB*i* statements associated with cursors are:

DBMS DECLARE *cursor_name* CURSOR FOR *sql_statement*

DBMS EXECUTE *cursor_name* [USING *var* [, *var*]]

DBMS CONTINUE [*cursor_name*]

DBMS CLOSE CURSOR *cursor_name*

To use a cursor, the above sequence of statements must be followed. Any number of EXECUTEs and CONTINUEs may be issued, perhaps with different variable names. A CONTINUE will always try to continue the last EXECUTE for that cursor name. A CONTINUE without a cursor name tries to continue the last non-cursor SELECT statement.

The **JAM** variables in a DBMS EXECUTE are the arguments for the variables in that cursor's associated SQL statement. The SQL variables are replaced by the value of the corresponding **JAM** variables, as determined by the order, going from left to right. Thus the first **JAM** variable corresponds to the first SQL variable, and so on. If there are not enough **JAM** arguments, an error is generated. Any extra arguments are ignored.

When an index is supplied for a **JAM** variable used in a DBMS EXECUTE, the corresponding element of that **JAM** array is used. If the array is word-wrapped, the value used is the value of all the elements concatenated together, starting from the given index, to the end of the array. If no index is supplied and the variable is an array, then the default index of "1" is assumed.

Using Multiple Cursors

A cursor allows an SQL statement to be precompiled, before actual execution. At compile time (DBMS DECLARE...CURSOR...) the

statement may use variables instead of constants. The values for the variables are supplied at the time of execution. For example, in the statement:

```
DBMS DECLARE nba2 CURSOR FOR \
insert into TEAMS (TEAM, CITY) values (::TEAM, ::CITY)
```

JPL will replace the double-colon with a single colon, which is the required prefix for variables. The insert gets compiled with 2 arguments (:TEAM and :CITY) and associated with the cursor nba2. It can be executed by simply supplying the **JAM** variable names which contain the values for :TEAM and :CITY. For example:

```
DBMS EXECUTE nba2 USING Team, City
```

where Team and City may be 2 fields. The insert statement can be executed a number of times, using different arguments, without redeclaring it, and thus save the compilation time.

A cursor name is just an identifier. Any set of non-blank characters are allowed in the name. The name may be as long as a **JAM** variable name.

Multiple Cursors and SELECT Statements

A SELECT statement may also be associated with a cursor. The additional advantage here is that with two SELECTs associated with two different cursors, the user can jump back and forth without having to re-issue the queries. For example, the following is a valid sequence of JPL statements:

```
DBMS DECLARE sel_nba CURSOR FOR SELECT * FROM NBATEAMS
```

```
DBMS DECLARE sel_nfl CURSOR FOR SELECT * FROM NFLTEAMS
DBMS EXECUTE sel_nba  (fetches 16 rows into form NBA)
DBMS EXECUTE sel_nfl  (fetches 16 rows into form NFL)
DBMS CONTINUE sel_nba   (fetches next 16 NBA rows)
DBMS CONTINUE sel_nfl  (fetches next 16 NFL rows)
```

Of course, a SELECT statement may also have variables in its WHERE clause, allowing slight modifications of the query with each DBMS EXECUTE. For example:

```
DBMS DECLARE nbateam CURSOR FOR \
SELECT * FROM NBATEAMS WHERE TEAM = ::teamname
DBMS EXECUTE nbateam USING NAME
DBMS EXECUTE nbateam USING TNAME
```

In the above example, "NAME" may be a **JAM** field, and TNAME a JPL variable.

Note that at the time of DECLAREing a SELECT cursor, **JAM/DB*i*** will map the columns of the target list into **JAM** fields or variables. Therefore, as far as the SELECT destination names are concerned, the contexts of the DECLARE and the EXECUTE should be equivalent. A safe way to do this would be to make the destinations data dictionary variables. Data dictionary items are available in every context, although their attributes may be over-ridden by local entities of the same name.

Cursor Management

There may be at most 9 cursors active at any time. This does not include the default cursors (see below). A cursor is active if it has been connected (DBMS CONNECT) and has not been closed (DBMS CLOSE CURSOR).

A cursor remains associated with a particular SQL statement until it is either closed, in which case the cursor name ceases to exist, or it is redeclared with another SQL statement. Closing a cursor frees some memory, so it may be useful to keep a minimum number of cursors active.

Upward Compatibility

All non-cursored JAM/DB*i* commands still behave as they used to. Ordinary SQL and DBMS statements may be freely mixed with the cursor commands. Non-cursor JAM/DB*i* commands do not allow arguments, and so may be a little quicker to execute. Other non-SELECT statements or cursored statements may be executed between a non-cursor SELECT and its CONTINUE. The CONTINUE will still try to fetch the next set of rows. For this reason, non-cursor statements may be thought of as using two default cursors, one for SELECT statements and one for non-SELECT statements. The only difference is that arguments are not allowed.

Transactions

Within a transaction, any changes to a table will lock all or portions of that table. This will prevent any other cursor (i.e., other than the one associated with the transaction) from accessing the table. For instance,

using just non-cursor commands inside a transaction, a SELECT will not be able to retrieve rows from a table being modified in that transaction.

II. Error Processing

There are 3 DBMS statements for handling errors and warnings:

DBMS ERROR_CONTINUE

DBMS ERROR [*number_var* [*message_var*]]

DBMS WARN [*warn-var*]

where:

number_var, *message_var* — are **JAM** variable or field identifiers. They may be array element identifiers, with one of the following form:

id[int] or *id[id]*

where *id* is a **JAM** identifier, and *int* is an integer.

The default action on any error, either **JAM/DB*i*** or SQL, is to display an error message, followed by the JPL statement that caused the error. When the two messages are acknowledged by hitting the space bar, **JAM/DB*i*** aborts the JPL procedure in which the error occurred. This conforms to the old behavior of version 3.16.

Issuing a DBMS ERROR_CONTINUE will prevent **JAM/DB*i*** from aborting the JPL procedure on an error. Error messages still get displayed as above.

Executing a DBMS ERROR with 1 or 2 JAM variable names will cause JAM/DB*i* to store the error number and message (if applicable) in the corresponding variables, after which JAM/DB*i* moves on to the next statement. A DBMS ERROR without any following variable names causes JAM/DB*i* to revert to default behavior.

Note that in the default mode only negative error codes, or those signalling actual errors, get displayed. However when error trapping is on, all errors and informational codes returned by the database get inserted into the appropriate JAM variables.

All internal (JAM/DB*i* or JAM) errors are always displayed at the bottom of the screen, followed whenever possible by the JPL statement during which the error occurred.

The behavior of DBMS WARN is described in the JAM/DB*i* 4.0 documentation.

Error Types

There are 3 types of errors that can occur while running JAM/DB*i*: SQL errors, JAM/DB*i* errors, and internal errors.

SQL Errors are errors reported by the database system. These usually indicate an error in the SQL statement or database access. SQL errors are displayed at the bottom of the screen by default. This behavior can be modified by the DBMS commands described above. An attempt is made to distinguish between actual *errors* and other informational messages. In most databases, errors have negative numeric codes and informational messages have positive numbers. Only *errors* get displayed on the screen by the default error handler. However error trapping will trap all errors and messages.

JAM/DB*i* *Errors* are errors in the JAM/DB*i* commands that are detected by JAM/DB*i*. An example is an attempt to use an undeclared cursor (see Chapter I). These errors always result in the JPL procedure being aborted and the error message and offending statement getting displayed on the screen. Error trapping will not modify this behavior. All such errors should have been caught at application development time.

Internal Errors are errors usually caused by an internal inconsistency in JAM/DB*i*. JAM/DB*i* handles these the same way it handles JAM/DB*i* errors.

Signalling End-of-SELECT

A database error code NO_MORE_ROWS (1403 in ORACLE) is signalled whenever an execution of a SELECT or a CONTINUE results in no new fetches. This error code will not be flashed at the bottom of the screen if default error processing is on. However, it may be trapped into a JAM variable (e.g., ERRCODE) using a statement like DBMS ERROR ERRCODE.

When a SELECT is being executed, the destination fields or variables are always cleared before performing any fetches. Thus an empty SELECT buffer will result in empty destination fields. On a CONTINUE however, if there is no data to be fetched, the destination fields or variables are not cleared.

DBMS START and Error Signalling

An argument greater than 1 in a DBMS START statement causes that number of rows of selected data to be ignored in a query. Any errors that internal fetches of those rows may cause are also ignored. This

means that if a DBMS START command causes a particular query to ignore all its selected rows, then the execution of that query will not cause any SQL errors to be signalled. End-of-SELECT messages are still available and can be trapped into a **JAM** variable as usual.

III. Text Datatype and Word-Wrapped Arrays

JAM variables have a length limit of 255 characters. Word-wrapped JAM array fields are used to handle data longer than that length. This is useful for handling TEXT database data types. JAM arrays that are not fields in the current form cannot be word-wrapped.

SELECTing into Word-Wrapped Arrays

If a word-wrapped array is one of the destinations for a SELECTed column, fetches are done one row at a time, with the word-wrap edit invoked on the relevant fields.

Updating from a Word-Wrapped Array

There are 2 ways to get the value of a full JAM array (i.e., all the elements) as 1 string into a JAM/DB*i* SQL statement. Let JA be an array:

```
SQL Insert into TABLE1 (LONGCOL) values (":JA")

DBMS DECLARE tbl1 CURSOR FOR \
SQL Insert into TABLE1 (LONGCOL) values (::JAVAL)
DBMS EXECUTE tbl1 USING JA
```

The first example uses colon expansion to get the value of the array. The second sequence of commands uses a cursor and SQL variables. In the second case, JA has to be a word-wrapped array.

Note that in the first example, the INSERT command is restricted in size by the JPL statement length limitations (approx. 2,000 characters). Therefore, when storing 'Long' values into a table, the second method is recommended.

IV. Customizing Query Result Destinations

The following commands specify where to send the results of a database query:

DBMS REDIRECT *cursor_name* TO *file_name* [TEE]

DBMS REDIRECT *cursor_name*

DBMS CATQUERY *cursor_name* TO *jam_fvar*

DBMS CATQUERY

DBMS OCCUR [*number_1* / *current*] [MAX *number_2*]

DBMS OCCUR

where:

cursor_name	is the name of a previously declared cursor (see Chapter I)
file_name	is the name of a file, including full path if not in current directory. There can be no embedded spaces in the file name.
jam_fvar	is the name of a JAM field or variable that will be active during execution of the intended query.
number_1 *number_2*	Integers greater than 0.

These commands allow a query's results to be sent to a file or a JAM variable, bypassing any column mapping. They also allow the user to specify a start index into the destination array and maximum number of rows to fetch.

In addition, query result column mapping now supports JPL variables as well as the other kinds of JAM variables.

Note: The term "fetch-execution" will be used to denote the execution of any of the following JAM/DB*i* statements.

SQL SELECT...
DBMS EXECUTE...
DBMS CONTINUE...

DBMS REDIRECT

JAM/DB*i* 4.8 includes a rudimentary report mechanism that allows the results of a query to be sent to a file. The command to specify this is associated with a cursor:

DBMS REDIRECT *cursor_name* TO *file_name* [TEE]

The optional TEE indicates that the query results should also go to its specified or default JAM variables.

If the query is going only to a file ("file-only mode"), or when the CATQUERY command is in effect (see below), the column widths used are derived from the table width definitions. If the results are also going to JAM fields (TEE), then the JAM widths are used. Columns in the file are separated by 2 spaces.

When the TEE option is being each fetch-execution of the query (using DBMS EXECUTE or DBMS CONTINUE) will send only the result rows fetched into the JAM variables or fields into the file. Several DBMS CONTINUEs may be needed to complete the report. In the file-only mode, just executing the cursor (DBMS EXECUTE) will send all the result rows into the named file.

The file *file_name* is opened when the REDIRECT command is executed, and all subsequent executions of the cursor add to the file. Any file of the same name that existed before the REDIRECT command is executed is over-written. The named file remains open, and associated with the specified cursor, until the cursor is either closed or redeclared, or the cursor is redirected to another file, or the following command is issued:

DBMS REDIRECT *cursor_name*

A number of files may be open at the same time, associated with different cursors, subject to machine limits. Redirects of a cursor not associated with a SELECT statement produce no results.

DBMS CATQUERY

Normally, the result columns of a SELECT statement get mapped to corresponding JAM variables or fields of the same name. The following command allows a full query result row to be fetched into a single JAM field or variable, by passing any default or specified individual column mappings:

DBMS CATQUERY *cursor_name* TO *jam_fvar*

After this command is executed all subsequent executions of a SELECT statement (DBMS EXECUTE, DBMS CONTINUE, SQL SELECT) will send their results to *jam_fvar*. This mode will remain in effect until the following command is executed:

DBMS CATQUERY *cursor_name*

A single fetch-execution will attempt to fill the destination array field or variable by returning as many rows as the array dimension. A DBMS CONTINUE will fetch the next batch. If the destination field is word-wrapped, only 1 row will be fetched per fetch-execution. Individual columns in the query result are separated by 2 spaces.

Note that *jam_fvar* must be accessible from the place that the DBMS CATQUERY is issued for it to work, otherwise a warning message is flashed on the screen and catquery mode for that cursor is reset. It is therefore advisable to put the CATQUERY destination into the data dictionary.

DBMS OCCUR

When the JAM destination for a query is an array or set of parallel arrays, the DBMS OCCUR command may be used to specify a part of the array to be used as the query destination. The default start index (the destination for the first fetched row of a fetch-execution) is 1. The default maximum number of rows to fetch in a particular fetch-execution equals the number of complete rows that the destination can hold (see your JAM/DB*i* manual). These 2 values can be modified by the following command:

DBMS OCCUR [*number_1* / *CURRENT*] [MAX *number_2*]

The first parameter in the OCCUR command is the start index. This may be a number (*number_1*), which will be the new start index, or CURRENT, in which case the start index will be whatever row the cursor is on at the time of the fetch-execution. The second parameter (MAX *number_2*) specifies the maximum number of rows to fetch. Both parameters are optional. However, if both are present, MAX *must* come second.

The new values of start index and MAX come into effect as soon as the DBMS OCCUR statement is executed. These values affect any subsequent fetch executions (including CONTINUE). To reset to default mode, use:

DBMS OCCUR

V. Miscellaneous

Suppressing Repeating Values in a Query

The following DBMS command will cause JAM/DB*i* to suppress repeating values in columns of a query result:

DBMS SUPREPS *int {, int}**

The integer arguments represent the column numbers, in any order, by position in a SELECT. The first column number is 1. When a * is used in a SELECT statement (e.g., SELECT * FROM...) the column numbering is according to the SELECT statements output. This order usually follows that in the table definition. Column suppression is an ON/OFF command, and takes effect from the next fetch-execution (including cursors and CONTINUEs). It may be combined with any of the query destination customizing commands of the previous chapter.

```
DBMS SUPREPS 1, 3

SQL SELECT city, team, venue FROM homesites
DBMS CONTINUE
      ...

DBMS SUPREPS
```

In the above example, the columns for `city` and `venue` will have their repeating values suppressed, producing an output like the following (if the table is already sorted on `city` as the major key and `venue` as the secondary key):

N.Y.C.	Knicks	Garden
	Rangers	
	Globetrotters	
	Mets	Shea Stadium
Boston	Celtics	Garden
E. Ruthfd	Nets	Byrne Arena
	Devils	
	Giants	Giant Stadium
	Jets	

A SUPREPS command over-rides all previous commands. A DBMS SUPREPS without any arguments (as in the last line of the example) resets and turns suppression off.

Any column numbers greater than 0 may be given as arguments. Only those numbers relevant to a particular query will be considered. For example, `DBMS SUPREPS 7, 1, 3` would produce the same result as above. If a subsequently issued query had a seventh column, that also would have been suppressed. At most, 25 columns can be targeted by a SUPREPS command.

Printing a File

The following DBMS command may be used to print a file:

DBMS PRINT *file_name*

where

file_name is the name of a file, including the full path if not in the current directory. There can be no embedded spaces in the file name.

The JAM configuration variable SMLPRINT should be set to the appropriate print command string (see your **JAM Configuration and Utilities** guide).

JAM Variables and JPL Variables

A JAM variable denotes any of the 3 kinds of variables supported by JAM:

- Local JPL variables;
- JAM screen fields or arrays;
- JAM data dictionary variables.

JAM allows the same name to have 3 different definitions in these 3 locations. The above sequence also gives the order of precedence in which the corresponding definitions take effect, with JPL variable definitions superseding the rest.

JAM/DB*i* now fully supports variable substitution from JPL variables for all SQL statements.

NULL Values

When writing into the database (e.g., INSERT, UPDATE), an empty string is interpreted as NULL. In the statement:

```
SQL INSERT INTO PLAYOFF (GAME_NBR, CITY, DATE) \
        VALUES ('', ':City', ':Date')
```

a NULL is being inserted into the integer field GAME_NBR. If the JAM fields City and Date are empty, those values will also be translated to NULL.

When executing a cursored statement, if any of the argument fields or variables are empty, they will be treated as NULLs.

Appendix B: Database-Specific Commands for ORACLE

NOTE: ORACLE converts column names and column alias names to upper case. When writing SQL SELECT statements, be sure that all JAM screen and data dictionary variables used as column destinations have UPPER CASE names, regardless of any column aliasing.

DBMS CATQUERY *cursor_name* [TO *jam_var*]

cursor_name is the identifier of an open cursor.

jam_var is the name of a JAM field or variable which will be active when the cursor is executed.

This command redirects the results of a query to a JAM variable, bypassing the normal JAM/DB*i* column mapping. The redirection remains in effect until you close the cursor, redeclare it, or execute DBMS CATQUERY *cursor_name* without the TO clause.

DBMS CLOSE CURSOR *cursor_name*

cursor_name is the identifier of an open cursor.

This command closes the specified cursor.

DBMS COMMIT

This command writes all transactions since the previous DBMS COMMIT (or DBMS LOGON) to the database.

DBMS CONTINUE [*cursor_name*]

cursor_name is the identifier of an open cursor.

This command fetches the next *n* rows of a query result into **JAM**, where *n* is the smallest number of array occurrences involved in the fetch.

DBMS CONTINUE with no argument continues the most recent query not associated with a cursor. DBMS CONTINUE *cursor_name* continues the query specified by *cursor_name*.

DBMS COUNT *count_var*

count_var is a **JAM** field or data dictionary variable.

This command returns the number of rows that were fetched into a **JAM** array and stores the result in *count_var*.

DBMS COUNT does not necessarily return the same value as ORACLE's COUNT function. The standard COUNT returns the number of values in a column which satisfy the SELECT conditions. DBMS COUNT, however, returns the number of rows fetched into **JAM**.

DBMS DECLARE *cursor_name* CURSOR FOR *sql_stmt*

cursor_name is an identifier for a cursor. It cannot contain blanks and can have the same maximum length as a JAM variable.

sql_stmt is an ORACLE SQL statement, which may contain variables.

This command precompiles *sql_stmt* before actual execution. The cursor remains open (i.e., the compiled *sql_stmt* remains available) until you execute a DBMS CLOSE CURSOR command.

DBMS ERROR [*code_var* [*message_var*]]

code_var and *message_var* are JAM variables or field identifiers. They may take any of the following forms:

id
id[int]
id[id]

where *id* is a JAM identifier and *int* is an integer. *id [int]* and *id[id]* are references to array elements.

This command causes JAM/DB*i* to store error codes and messages in the specified variables or array elements. Error trapping remains in effect until you execute the command DBMS ERROR with no arguments.

DBMS ERROR_CONTINUE

This command prevents **JAM/DB*i*** from aborting a JPL procedure when an error is detected. DBMS ERROR_CONTINUE remains in effect until you execute a DBMS ERROR command.

DBMS EXECUTE *cursor_name* [USING *var1* [, *var2*...]]

cursor_name is the identifier for an open cursor.

var1, *var2* ... *varn* are **JAM** variable references. They may take any of the following forms:

id
id[int]
id[id]

where *id* is a **JAM** identifier and *int* is an integer. *id [int]* and *id[id]* are references to array elements.

This command executes the SQL statement specified in the corresponding DBMS DECLARE command. If the SQL statement contains variables, **JAM/DB*i*** substitutes the values of *var1*, *var2*, ... *varn*.

DBMS LOGOFF

This command closes an ORACLE session.

DBMS LOGON *user_id password* | *user_id/password*

This command opens an ORACLE session.

DBMS OCCUR [*int1* | CURRENT] [MAX *int2*]

int1 and *int2* are integers greater than 0.

When the destination of a query is an array or set of parallel arrays, the default destination for the first row of a query result is the first row of the array. The maximum number of rows returned by a single fetch is the total number of rows in the array. DBMS OCCUR allows you to change these defaults.

DBMS OCCUR *int1* specifies that *int1* is the first row of the array to be filled.

DBMS OCCUR CURRENT specifies that the array row where JAM's cursor is located is the first row of the array to be filled.

DBMS OCCUR MAX *int2* specifies that *int2* is the maximum number of rows to be fetched.

If you specify both a starting row and a maximum number of rows, MAX *int2* must be the second clause in the command. The destination parameters you specify remain in effect until you execute another DBMS OCCUR command. Executing DBMS OCCUR with no arguments restores the default destination parameters.

DBMS PRINT *file_name*

file_name is the name of an existing file. You must include the full path for files outside the current directory.

This command will print the contents of the file. You must have the SMLPRINT **JAM** configuration variable set to the appropriate command string (see **JAM Configuration and Utilities Guide**).

DBMS REDIRECT *cursor_name* [TO *file_name* [TEE]]

cursor_name is the identifier of an open cursor.

file_name is the name of a file which DBMS REDIRECT will open. You must include the full path for files outside the current directory.

This command redirects the results of a query to a file. Executing DBMS REDIRECT opens the file. If the file already exists, all previous data will be over-written. Subsequent executions of the cursor append query output to the file. The redirection remains in effect until you close the cursor, redeclare it, or execute the command DBMS REDIRECT *cursor_name* without the TO clause.

DBMS REDIRECT *cursor_name* TO *file_name* TEE directs the query results to both *file_name* and other specified or default **JAM** variables. If you do not use the TEE option, query results will go only to *file_name*.

DBMS ROLLBACK

This command flushes all transactions since the last DBMS COMMIT or DBMS LOGON without writing them to the database.

DBMS START *row_number*

row_number is a positive integer.

This command causes JAM/DB*i* to read and ignore a specified number records (*row_number* - 1) before fetching the remaining query results into JAM.

DBMS SUPREPS *int1* [, *int2* ... *int25*]

int1, int2, ... int25 are integer references to columns in a SELECT statement. Thus, DBMS SUPREPS 1, 3 refers to the first and third columns in a subsequent SELECT.

This command suppresses repeating values in the column when the data are fetched into a JAM array. Suppression of repeated values remains in effect until you execute a DBMS SUPREPS with no arguments.

DBMS WARN *warn_array*

warn_array is the identifier of an 8-element JAM data dictionary array.

This command causes JAM/DB*i* to place a "W" in each element of *warn_array* that matches a database warning. The ORACLE interface defines the elements as follows:

Element	Meaning
1	There was a warning issued
2	One or more fields was truncated on output
3	A null value was ignored in a function call
4	One or more output fields in a SELECT was not in the JAM data dictionary
5	UPDATE or DELETE did not have a WHERE clause
6	unused
7	Implicit rollback (e.g., a row was locked)
8	Row changed between SELECT and fetch of the row

New Features in JAM/DBi for SYBASE / SQL Server Release 4.8

Contents

I. Multiple Cursors

In the following,

cursor_name	is an identifier, consisting of non-blank characters, with maximum same as a JAM variable.
database_name	is the name of the database that the cursor is logging into
sql_statement	is an SQL statement, possibly containing variables (syntax is database specific).

The JAM/DB*i* statements associated with cursors are:

DBMS CONNECT *cursor_name* TO DATABASE *database_name*

This command activates the cursor. It must be issued *before* any other commands which refer to the cursor.

DBMS DECLARE *cursor_name* CURSOR FOR *sql_statement*

This command defines the cursor. It cannot be issued unless *cursor_name* is active (i.e., DBMS CONNECT *cursor_name*... has been issued and DBMS CLOSE CURSOR *cursor_name* has *not* been issued).

DBMS EXECUTE *cursor_name*

This command cannot be issued unless *cursor_name* is active and defined (i.e., CONNECT *cursor_name* and DECLARE *cursor_name* have been issued; CLOSE CURSOR *cursor_name* has *not* been issued).

DBMS CONTINUE [*cursor_name*]

This command continues a preceding EXECUTE. It cannot be issued for a named cursor unless the *cursor_name* is active, defined, and has been executed. CONTINUE *cursor_name* can be re-issued any number of times, so long as DBMS CLOSE CURSOR *cursor_name* has not been issued. A CONTINUE without a cursor name tries to continue the last non-cursor SELECT.

DBMS CLOSE CURSOR *cursor_name*

This command inactivates the named cursor.

Multiple Cursors and SELECT Statements

A SELECT statement may also be associated with a cursor. The additional advantage here is that with two SELECTs associated with two different cursors, the user can jump back and forth without having to re-issue the queries. For example, the following is a valid sequence of JPL statements:

```
DBMS DECLARE sel_nba CURSOR FOR SELECT * FROM NBATEAMS
DBMS DECLARE sel_nfl CURSOR FOR SELECT * FROM NFLTEAMS
DBMS EXECUTE sel_nba      (fetches 16 rows into form NBA)
DBMS EXECUTE sel_nfl      (fetches 16 rows into form NFL)
DBMS CONTINUE sel_nba     (fetches next 16 NBA rows)
DBMS CONTINUE sel_nfl     (fetches next 16 NFL rows)
```

Cursor Management

There may be at most 9 cursors active at any time. This does not include the default cursors (see below). A cursor is active if it has been connected (DBMS CONNECT) and has not been closed (DBMS CLOSE CURSOR). A cursor remains associated with a particular SQL statement until it is either closed, in which case the cursor name ceases to exist, or it is redeclared with another SQL statement. Closing a cursor frees some memory, so it may be useful to keep a minimum number of cursors active.

Upward Compatibility

All non-cursored JAM/DB*i* commands still behave as they used to. Ordinary SQL and DBMS statements may be freely mixed with the cursor commands. Non-cursor JAM/DB*i* commands do not allow arguments, and so may be a little quicker to execute. Other non-SELECT statements or cursored statements may be executed between a non-cursor SELECT and its CONTINUE. The CONTINUE will still try to fetch the next set of rows. For this reason, non-cursor statements may be thought of as using two default cursors, one for SELECT statements and one for non-SELECT statements. The only difference is that arguments are not allowed.

Transactions

Within a transaction, any changes to a table will lock all or portions of that table. This will prevent any other cursor (i.e., other than the one associated with the transaction) from accessing the table. For instance, using just non-cursor commands inside a transaction, a SELECT will not be able to retrieve rows from a table being modified in that transaction.

Execution of select and non-select statements without named cursors (i.e., using the default cursor) actually activates two independent cursors. Therefore, any transaction which includes both select and non-select statements should use named cursors.

II. Error Processing

There are 2 DBMS statements for handling errors:

DBMS ERROR_CONTINUE

DBMS ERROR [*number_var* [*message_var*]]

where:

number_var, *message_var* — are **JAM** variable or field identifiers. They may be array element identifiers, with one of the following form:

id or *id[int]* or *id[id]*

where *id* is a **JAM** identifier, and *int* is an integer.

The default action on any error, either **JAM/DB*i*** or SQL, is to display an error message, followed by the JPL statement that caused the error. When the two messages are acknowledged by hitting the space bar, **JAM/DB*i*** aborts the JPL procedure in which the error occurred. This conforms to the old behavior of version 3.16

Issuing a DBMS ERROR_CONTINUE will prevent **JAM/DB*i*** from aborting the JPL procedure on an error. Error messages will still be displayed as above.

Executing a DBMS ERROR with 1 or 2 **JAM** variable names will cause **JAM/DB*i*** to store the error number and message (if applicable) in the corresponding variables, after which **JAM/DB*i*** moves on to the next statement. A DBMS ERROR without any following variable names causes **JAM/DB*i*** to revert to default behavior.

Note that in the default mode only negative error codes, or those signalling actual errors, are displayed. However when error trapping is on, all errors and informational codes returned by the database are inserted into the appropriate JAM variables.

All internal (JAM/DB*i* or JAM) errors are always displayed at the bottom of the screen, followed whenever possible by the JPL statement during which the error occurred.

Error Types

There are 3 types of errors that can occur while running JAM/DB*i*: SQL errors, JAM/DB*i* errors, and internal errors.

SQL Errors are errors reported by the database system. These usually indicate an error in the SQL statement or database access. SQL errors are displayed at the bottom of the screen by default. This behavior can be modified by the DBMS commands described above. An attempt is made to distinguish between actual *errors* and other informational messages. In most databases, errors have negative numeric codes and informational messages have positive numbers. Only *errors* are displayed on the screen by the default error handler. However error trapping will trap all errors and messages.

JAM/DB*i* *Errors* are errors in the JAM/DB*i* commands that are detected by JAM/DB*i*. An example is an attempt to use an undeclared cursor (see Chapter I). These errors always result in the JPL procedure being aborted and the error message and offending statement being displayed on the screen. Error trapping will not modify this behavior. All such errors should have been caught at application development time.

Internal Errors are errors usually caused by an internal inconsistency in JAM/DB*i*. JAM/DB*i* handles these the same way it handles JAM/DB*i* errors.

End-of-SELECT Signal

An end-of-SELECT signal is not displayed if default error processing is on. However, if error trapping is on, an error code of 100, severity 0 will be trapped in the specified variable. For example:

```
DBMS ERROR errcode
cat errcode "0"
SQL SELECT * from TABLEA
if (errcode == "0")
        msg emsg "Hit F2 for more rows."
else if errcode == "000100:000000"
        msg emsg "Done"
else
        msg emsg "Error"
```

DBMS START and Error Signalling

An argument greater than 1 in a DBMS START statement causes one fewer than that number of rows of selected data to be ignored in a query. Any errors that internal fetches of those rows may cause are also ignored. This means that if a DBMS START command causes a particular query to ignore all its selected rows, then the execution of that query will not cause any SQL errors to be signalled. End-of-SELECT messages are still available and can be trapped into a **JAM** variable as usual.

III. Text Datatype and Word-Wrapped Arrays

JAM variables have a length limit of 255 characters. Word-wrapped JAM array fields are used to handle data longer than that length. This is useful for handling TEXT database data types. JAM arrays that are not fields in the current form will not be treated having word-wrapped edit.

SELECTing into Word-Wrapped Arrays

If a word-wrapped array is one of the destinations for a SELECTed column, fetches are done one row at a time, with the word-wrap edit invoked on the relevant fields.

Updating from a Word-Wrapped Array

There are 2 ways to get the value of a full JAM array (i.e., all the elements) as 1 string into a JAM/DB*i* SQL statement. Let JA be an array:

```
SQL Insert into TABLE1 (LONGCOL) values (":JA")

SQL Insert into TABLE1 (LONGCOL) values (::JA)
```

The first example uses colon expansion to get the value of the array. In the second case, JA has to be a word-wrapped array.

Note that in the first example, the INSERT command is restricted in size by the JPL statement length limitations (approx. 2,000 characters). Therefore, when storing 'Long' values into a table, the second method is recommended.

IV. Customizing Query Result Destinations

The following commands specify where to send the results of a database query:

DBMS REDIRECT *cursor_name* TO *file_name* [TEE]

DBMS REDIRECT *cursor_name*

DBMS CATQUERY *cursor_name* TO *jam_fvar*

DBMS CATQUERY

DBMS OCCUR [*number_1* / *current*] [MAX *number_2*]

DBMS OCCUR

where:

cursor_name	is the name of a previously declared cursor (see Chapter I)
file_name	is the name of a file, including full path if not in current directory. There can be no embedded spaces in the file name.
jam_fvar	is the name of a JAM field or variable that will be active during execution of the intended query.
number_1 *number_2*	Integers greater than 0.

These commands allow a query's results to be sent to a file or a JAM variable, bypassing any column mapping. They also allow the user to specify a start index into the destination array and maximum number of rows to fetch.

In addition, query result column mapping now supports JPL variables as well as the other kinds of JAM variables.

Note: The term "fetch-execution" will be used to denote the execution of any of the following JAM/DB*i* statements.

SQL SELECT...
DBMS EXECUTE...
DBMS CONTINUE...

DBMS Redirect

JAM/DB*i* 4.7 includes a rudimentary report mechanism that allows the results of a query to be sent to a file. The command to specify this is associated with a cursor:

DBMS REDIRECT *cursor_name* TO *file_name* [TEE]

The optional TEE indicates that the query results should also go to its specified or default JAM variables.

If the query is going only to a file ("file-only mode"), or when the CATQUERY command is in effect (see below), the column widths used are derived from the table width definitions. If the results are also going to JAM fields (TEE), then the JAM widths are used. Columns in the file are separated by 2 spaces.

When the TEE option is present each fetch-execution of the query (using DBMS EXECUTE or DBMS CONTINUE) will send only the result rows fetched into the JAM variables or fields into the file. Several DBMS CONTINUEs may be needed to complete the report. In the file-only mode, just executing the cursor (DBMS EXECUTE) will send all the result rows into the named file.

The file *file_name* is opened when the REDIRECT command is executed, and all subsequent executions of the cursor add to the file. Any file of the same name that existed before the REDIRECT command is executed is over-written. The named file remains open, and associated with the specified cursor, until the

cursor is either closed or redeclared, or the cursor is redirected to another file, or the following command is issued:

DBMS REDIRECT *cursor_name*

A number of files may be open at the same time, associated with different cursors, subject to machine limits. Redirects of a cursor not associated with a SELECT statement produce no results.

DBMS Catquery

Normally, the result columns of a SELECT statement get mapped to corresponding JAM variables or fields of the same name. The following command allows a full query result row to be fetched into a single JAM field or variable, by passing any default or specified individual column mappings:

DBMS CATQUERY *cursor_name* TO *jam_fvar*

After this command is executed all subsequent executions of a SELECT statement (DBMS EXECUTE, DBMS CONTINUE, SQL SELECT) will send their results to *jam_fvar*. This mode will remain in effect until the following command is executed:

DBMS CATQUERY *cursor_name*

A single fetch-execution will attempt to fill the destination array field or variable by returning as many rows as the array dimension. A DBMS CONTINUE will fetch the next batch. If the destination field is word-wrapped, only 1 row will be fetched per fetch-execution. Individual columns in the query result are separated by 2 spaces.

Note that *jam_fvar* must be accessible from the place that the DBMS CATQUERY is issued for it to work, otherwise a warning message is flashed on the screen and catquery mode for that cursor is reset. It is therefore advisable to put the CATQUERY destination into the data dictionary.

DBMS Occur

When the JAM destination for a query is an array or set of parallel arrays, the DBMS OCCUR command may be used to specify a part of the array to be used as the query destination. The default start index (the destination for the first fetched row of a fetch-execution) is 1. The default maximum number of rows to fetch in a particular fetch-execution equals the number of complete rows that the destination can hold (see your JAM/DB*i* manual). These 2 values can be modified by the following command:

DBMS OCCUR [*number_1* / *CURRENT*] [MAX *number_2*]

The first parameter in the OCCUR command is the start index. This may be a number (*number_1*), which will be the new start index, or CURRENT, in which case the start index will be whatever row the cursor is on at the time of the fetch-execution. The second parameter (MAX *number_2*) specifies the maximum number of rows to fetch. Both parameters are optional. However, if both are present, MAX *must* come second.

The new values of start index and MAX come into effect as soon as the DBMS OCCUR statement is executed. These values affect any subsequent fetch executions (including CONTINUE). To reset to default mode, use:

DBMS OCCUR

V. Miscellaneous

Suppressing Repeating Values in a Query

The following DBMS command will cause JAM/DB*i* to suppress repeating values in columns of a query result:

DBMS SUPREPS *int {, int}**

The integer arguments represent the column numbers, in any order, by position in a SELECT. The first column number is 1. When a * is used in a SELECT statement (e.g., SELECT * FROM...) the column numbering is according to the SELECT statements output. This order usually follows that in the table definition. Column suppression is an ON/OFF command, and takes effect from the next fetch-execution (including cursors and CONTINUEs). It may be combined with any of the query destination customizing commands of the previous chapter.

```
DBMS SUPREPS 1, 3

SQL SELECT city, team, venue FROM homesites
DBMS CONTINUE
      ...

DBMS SUPREPS
```

In the above example, the columns for `city` and `venue` will have their repeating values suppressed, producing an output like the following (if the table is already sorted on `city` as the major key and `venue` as the secondary key):

N.Y.C.	Knicks	Garden
	Rangers	
	Globetrotters	
	Mets	Shea Stadium
Boston	Celtics	Garden

E. Ruthfd	Nets	Byrne Arena
	Devils	
	Giants	Giant Stadium
	Jets	

A SUPREPS command over-rides all previous commands. A DBMS SUPREPS without any arguments (as in the last line of the example) resets and turns suppression off.

Any column numbers greater than 0 may be given as arguments. Only those numbers relevant to a particular query will be considered. For example, `DBMS SUPREPS 7, 1, 3` would produce the same result as above. If a subsequently issued query had a seventh column, that also would have been suppressed. At most, 25 columns can be targeted by a SUPREPS command.

Printing a File

The following DBMS command may be used to print a file:

DBMS PRINT *file_name*

where

file_name is the name of a file, including the full path if not in the current directory. There can be no embedded spaces in the file name.

The JAM configuration variable SMLPRINT should be set to the appropriate print command string (see your **JAM Configuration and Utilities** guide).

JAM Variables and JPL Variables

A JAM variable denotes any of the 3 kinds of variables supported by JAM:

- Local JPL variables;
- JAM screen fields or arrays;
- JAM data dictionary variables.

JAM allows the same name to have 3 different definitions in these 3 locations. The above sequence also gives the order of precedence in which the corresponding definitions take effect, with JPL variable definitions superceding the rest.

JAM/DB*i* now fully supports variable substitution from JPL variables for all SQL statements.

NULL Values

When writing into the database (e.g., INSERT, UPDATE), an empty string will not be interpreted as NULL. In order to enter NULL values into a table, the word NULL must be specified. For instance,

```
SQL INSERT INTO PLAYOFF (TEAM, CITY, DATE) \
      VALUES (NULL, ':City', ':Date')
```

a NULL is being inserted into the character field TEAM. If the JAM fields `City` and `Date` are empty, those values will be translated as empty strings.

When a return value from a query is NULL, the string 'NULL' will be displayed in the corresponding JAM variable.

Scrolling Through the SELECTed Rows

Scrolling is allowed unless a buffer has been setup to store the rows which are already fetched. Use the following command to set a buffer before executing the query.

DBMS SET_BUF *number of rows to allow in buffer* [*cursor_name*]

Buffering takes up memory, so set a buffer only when CONTINUE_UP, CONTINUE_TOP, or CONTINUE_BOT is desired. Reset the buffer as soon as it is no longer necessary. Buffering will be reset by passing integer value zero as argument to DBMS SET_BUF.

The following commands scroll through the selected set of rows:

DBMS CONTINUE [*cursor_name*]

DBMS CONTINUE_DOWN [*cursor_name*]

DBMS CONTINUE_UP [*cursor_name*]

DBMS CONTINUE_TOP [*cursor_name*]

DBMS CONTINUE_BOT [*cursor_name*]

The first two are equivalent, and try to fetch the next page of rows into JAM's current view window (e.g., an on-screen array field). See description of SELECT for determining number of rows comprising a view window.

CONTINUE_UP will try to fetch the previous page of rows.

CONTINUE_TOP displays the first page of rows.

CONTINUE_BOT displays the last page of rows.

The following example shows the sequence of command to be issued:

```
DBMS SET_BUF 100

SQL SELECT city, team, venue FROM homesites
DBMS CONTINUE_UP
      ...

DBMS SET_BUF 0
```

Normally, there is no overlap between the rows displayed (i.e., fetched into the **JAM** destination) before and after the CONTINUE_UP (or CONTINUE_DOWN) command is issued. However, when that command positions **JAM**'s current view window at the top (bottom) of the select set, the first (last) row of the set is fetched into the first (last) element of the destination; the rest of the rows fill up the rest of the destination. When the number of rows in the select set is less than the size of the view window, the elements are filled from top down.

If the buffer set up is not large enough to hold all return rows, the top of the buffer will be cleared as newly fetched rows are stored. In that case, CONTINUE_TOP will not return the first selected rows, but the first row in the row buffer. For instance, if the row buffer is of size 100, and 120 rows are fetched, then CONTINUE_TOP will starting fetching from row 21 instead of the first row (which has been cleared).

The DBMS COUNT... command can be used to determine the number of rows fetched after each CONTINUE_*.

If the column repeat suppression mode is on (see DBMS SUPREPS), scrolling up or to the top or bottom will always display all the columns of the row in the first element, suppression always being executed reading from the top down.

Selecting BINARY Datatypes

Binary data cannot be retrieved into JAM variables or fields. The JAM/DB*i* library (`sybdbi.a`) contains a global variable named `DBi_image`, of type DBBINARY*, which will point to the retrieved data (DBBINARY is a Sybase DB-Library datatype). This variable may be freed after being processed; if not, it will not be freed until the next binary column selection is invoked. Only 1 row will be retrieved if the select statement involves a column of binary type. No matter how many binary fields are selected, only the first binary column will be retrieved.

Browse Mode

(Browse Mode is not available for servers running SYBASE Ver 3.X)

JAM/DB*i* supports the BROWSE MODE of SYBASE DB-Library by supplying the following commands:

DBMS BROWSE *select_statement*

DBMS UPDATE *cursor_name* SET *col1 = exp1 [, col2 = exp2...]*

These allow browsing selected rows and updating their values one row at a time. For example,

```
DBMS COUNT row
DBMS  BROWSE  select  field1  =  PRODUCT.PRICE  from\
      CLIENT, POSITIONS where PRODUCT.PRICE\
      = POSITIONS.PRICE.
while (row > 0)
{
      DBMS UPDATE PRODUCT set PRICE = :field1 + 2
      DBMS CONTINUE
}
```

Handling Stored Procedures and Their Results

```
DBMS PROC [cursor_name] EXEC\
      <batch command for stored procedure>

DBMS CONTINUE [cursor_name]

DBMS NEXT [cursor_name]

DBMS CANCEL [cursor_name]

DBMS FLUSH [cursor_name]

DBMS RETURN var
```
(DBMS RETURN is available only for servers running SYBASE Ver 4.X)

There may be multiple SQL statements in a stored procedure. All SQL statements will work the same way as normal JPL SQL statements. Data returned via select statements will be mapped to JAM variables. If the number of rows of fetched data is beyond the size of the destination, the stored procedure will be suspended until you call either,

`DBMS CONTINUE`	to continue fetching data, or
`DBMS NEXT`	to flush the pending query and start executing the next command, or
`DBMS CANCEL`	to cancel the stored procedure:

If, however, there is no need to call DBMS CONTINUE, JAM/DB*i* will immediately continue to execute the next SQL statement in the stored procedure without waiting for a DBMS NEXT. A pending stored procedure will automatically be cancelled by any DBMS or SQL statement (other than DBMS CONTINUE, DBMS FLUSH, or DBMS NEXT) that uses the same cursor.

There are several ways to execute a stored procedure:

In the following example, the stored procedure `proc1` will be executed using the cursor `cursor_a`:

```
DBMS PROC cursor_a EXEC proc1
```

In the following example, stored procedure `proc2` will be executed using default cursor and the return status of the `proc2` will be trapped in the JAM variable `status`:

```
DBMS RETURN status
DBMS PROC EXEC proc2
```

(DBMS RETURN is only available for servers running SYBASE Ver 4.X)

Return-value parameters are supported only by SYBASE Ver 4.X servers. In the following example, the stored procedure `proc3` will receive parameters 1, 2, and 50; the return parameter of the stored procedure will be trapped in the JAM variable `result`:

```
DBMS PROC EXEC declare @const int\
select @const = 50\
exec proc3 1, 2, @result = @const output
```

`DBMS NEXT [cursor_name]`

> *cursor_name* is the identifier of an open cursor. If it is null, the default cursor will be used.
>
> This command flushes the select statement pending in the stored procedure, and then executes the next SQL statement in the stored procedure.

`DBMS CANCEL [cursor_name]`

cursor_name is the identifier of an open cursor. If it is null, the default cursor will be used.

This command cancels a suspended stored procedure.

`DBMS FLUSH [cursor_name]`

cursor_name is the identifier of an open cursor. If it is null, the default cursor will be used.

This command flushes any rows pending in the stored procedure or SQL SELECT statement associated with the given cursor. It is typically used when the last statement in a stored procedure is SELECT.

NOTE: No return values or return parameters are set until the stored procedure terminates. In particular, if any selected rows are pending, the return value and parameters will be unavailable. If the number of rows fetched by SELECT or DBMS CONTINUE exactly matches the number of elements in the destination variables, the SELECT is considered to be pending, since the "no more rows" condition cannot be detected until the next fetch. Use DBMS FLUSH (or DBMS NEXT, or DBMS CONTINUE) to ensure that no SELECT is pending.

Default Cursors

In JAM/DB*i* for SYBASE, a user is automatically logged on to the server with two cursors (DBPROCESSes) so that a query (a JPL SQL select statement) can be interrupted by other non-select JPL SQL statements. There is an option for the DBMS LOGON command, -C<num>, which changes this default feature. When <num> is 1, only one default cursor will be used when logging on. Using -C<num> with any number other than 1 will assume two cursors; calling DBMS LOGON without the -C option will do the same.

If a user is logged on with only *one* cursor, he may *not* use certain JAM/DB*i* features:

- DBMS BROWSE and DBMS UPDATE command will be disabled.
- Whenever a query is interrupted, it will not be resumed.

However, the above features can be implanted with two named cursors.

Appendix B: Database-Specific Commands for SYBASE / SQL Server

Note: This appendix summarizes the database specific commands for the BETA release of JAM/DB*i* version 4.7 for SYBASE / SQL Server.

DBMS BEGIN [*transaction_name* [*cursor_name*]]

transaction_name is the name assigned to a transaction.

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command marks the starting point of a transaction, and must be followed by a DBMS COMMIT or DBMS ROLLBACK.

DBMS BROWSE *query*
(DBMS BROWSE is not available if server is running SYBASE Ver 3.X)

query is a select statement. See the Sybase command reference for the correct syntax for select statements.

This command uses the default cursor to perform a browse mode select. The browse mode select is similar to an ordinary select except that JAM/DB*i* only fetches 1 row at a time. The DBMS UPDATE command may be used to update the current row. DBMS CONTINUE will fetch the next row.

DBMS CATQUERY *cursor_name* [TO *jam_var*]

cursor_name is the identifier of an open cursor.

jam_var is the name of a **JAM** variable.

This command redirects the results of a query to a **JAM** variable, bypassing the normal **JAM/DB*i*** column mapping. The redirection remains in effect until you close the cursor, redeclare it, or execute DBMS CATQUERY *cursor_name* without the TO clause.

DBMS CLOSE CURSOR *cursor_name*

cursor_name is the identifier of an open cursor.

This command closes the specified cursor.

DBMS COMMIT [*transaction_name* [*cursor_name*]]

transaction_name is the name assigned to a transaction.

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command marks the end point of a transaction. It must be used following a DBMS BEGIN.

DBMS CONNECT *cursor_name* TO DATABASE *database_name*

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

database_name is the name of the database which the cursor will log onto.

This command create a login (with the login attributes used in DBMS LOGON) to the server using the specified database.

DBMS CONTINUE [*cursor_name*]

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command fetches the next page of rows.

DBMS CONTINUE_BOT [*cursor_name*]

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command fetches the last page of rows.

DBMS CONTINUE_DOWN [*cursor_name*]

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command fetches the next page of rows.

DBMS CONTINUE_TOP [*cursor_name*]

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command fetches the first page of rows.

DBMS CONTINUE_UP [*cursor_name*]

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command fetches the previous page of rows.

DBMS COUNT *count_var*

count_var is a **JAM** variable.

This command sets *count_var* to record the number of rows fetched.

DBMS CREATE_PROC [*cursor_name*] EXEC *<command to create stored procedure>*

This command creates a stored procedure.

DBMS CREATE_TRIGGER [*cursor_name*] EXEC *<command to create trigger for stored procedure>*

This command creates the trigger for a stored procedure.

DBMS DECLARE *cursor_name* CURSOR FOR *sql_stmt*

cursor_name is an identifier for a cursor.

sql_stmt is a Sybase SQL statement.

This command stores *sql_stmt* before actual execution.

DBMS DROP_PROC [*cursor_name*] EXEC *procedure_name1* [, *procedure_name 2*,...]

This command removes a stored procedure.

DBMS DROP_TRIGGER *trigger_name1* [, *trigger_name2*,...]

This command removes the trigger for a stored procedure.

DBMS ERROR [*code_var* [*message_var*]]

code_var and *message_var* are **JAM** variables. They may take any of the following forms:

id
id[int]
id[id]

where *id* is a **JAM** identifier and *int* is an integer. *id [int]* and *id[id]* are references to array elements.

This command causes **JAM/DB*i*** to store error codes and messages in the specified variables or array elements. Error trapping remains in effect until you execute the command DBMS ERROR with no arguments.

JAM/DB*i* will assign *code_var* a code which consists of two 6-digit fields separated by a colon (i.e., xxxxxx:yyyyyy). The first field is either the Database Library error code or the DataServer error code. When error trapping is on, all error and informational (severity=0) messages are trapped into *code_var* and *message_var*.

DBMS ERROR_CONTINUE

This command prevents JAM/DB*i* from aborting a JPL procedure when an error is detected.

DBMS EXECUTE *cursor_name*

cursor_name is the identifier of an open cursor.

This command executes the SQL statement specified in the corresponding DBMS DECLARE command.

DBMS LOGOFF

This command exits from Sybase server.

DBMS LOGON *[-t timeout] [-H hostname] [-U username] [-P password] [-I interface] [-S server] [-C 1 or 2]*

-t timeout Number of seconds that DB-Library waits for a login response before timing out. A timeout value of 0 represents an infinite timeout period. The default is 60 seconds.

-H hostname Allows the user to specify a host name, changing the value in the dynamic system table `sysprocesses`, if logging from a different

computer than usual. If no host name is specified, the current computer name is assumed.

-U username Allows the user to specify a login name. Logins are case sensitive.

-P password Allows the user to specify a password. Passwords are case sensitive.

-I interface Allows the user to specify the name and location of the interfaces file that is searched as part of the process of connecting to SQL Server. The named file contains the name and network address of every available SQL Server on the network. If this option is not used, `isql` looks for a file named `interfaces`.

-S server Allows the user to specify the name of the particular SQL Server with which to connect. This is the name that SQL Server looks up in the `interfaces` file.

-C [1 or 2] Allows the user to specify the number of default cursors. The default is 2.

Note: If a flag is given without a parameter value, the value is taken as NULL, or 0 for timeout.

This command connects the Sybase server.

DBMS OCCUR [*int1* | CURRENT] [MAX *int2*]

int1 and *int2* are integers greater than 0.

DBMS OCCUR *int1* specifies that *int1* is the first row of the array to be filled.

DBMS OCCUR CURRENT specifies that the array row where **JAM**'s cursor is located is the first row of the array to be filled.

DBMS OCCUR MAX *int2* specifies that *int2* is the maximum number of rows to be fetched.

DBMS PRINT *file_name*

file_name is the name of an existing file.

This command will print the contents of the file. You must have the SMLPRINT **JAM** configuration variable set to the appropriate command string (see **JAM Configuration and Utilities Guide**).

DBMS PROC [*cursor_name*] EXEC *cmd*

cursor_name is the identifier of an open cursor.

cmd is the batch command which executes a stored procedure.

This command executes a stored procedure and does column mapping to JAM if there is a query within the stored procedure.

DBMS REDIRECT *cursor_name* [TO *file_name* [TEE]]

cursor_name is the identifier of an open cursor.

file_name is the name of a file which DBMS REDIRECT will open.

This command redirects the results of a query to a file. Executing DBMS REDIRECT opens the file. If the file already exists, all previous data will be over-written. Subsequent executions of the cursor append query output to the file. The redirection remains in effect until you close the cursor, redeclare it, or execute the command DBMS REDIRECT *cursor_name* without the TO clause.

DBMS REDIRECT *cursor_name* TO *file_name* TEE directs the query results to both *file_name* and other specified or default **JAM** variables. If you do not use the TEE option, query results will go only to *file_name*.

DBMS RETURN *return_status_var*
(DBMS RETURN is available only if server is running SYBASE Ver 4.X)

return_status_var is a **JAM** variable.

This command set *return_status_var* as the destination for the return status of a stored procedure.

DBMS ROLLBACK [*transaction_name* / *savepoint_name* [*cursor_name*]]

transaction_name is the name assigned to a transaction.

savepoint_name is the name assigned to the savepoint of a transaction.

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command flushes all transactions since the last DBMS BEGIN or savepoint.

DBMS SAVE *savepoint_name* [*cursor_name*]

savepoint_name is the name assigned to the savepoint of a transaction.

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

This command sets a savepoint within a transaction.

DBMS SET_BUF *integer* [*cursor_name*]

integer is an integer greater than or equal to 0.

cursor_name is the identifier of an open cursor. If it is null, default cursor will be used.

DBMS SET_BUF with a non-zero number will set the DBBUFFER option with the specified number of rows. This option must be set for DBMS CONTINUE_UP, DBMS CONTINUE_TOP, and DBMS CONTINUE_BOT commands to function.

DBMS SET_BUF with *integer* = 0 resets the buffer.

DBMS START *row_number*

row_number is a positive integer.

This command causes JAM/DB*i* to read and ignore a specified number of records (*row_number* - 1) before fetching the remaining query results into JAM.

DBMS SUPREPS *int1* [, *int2* ... *int25*]

int1, *int2*, ... *int25* are integer references to columns in a SELECT statement. Thus, DBMS SUPREPS 1, 3 refers to the first and third columns in a subsequent SELECT.

This command suppresses repeating values in the column when the data are fetched into a JAM array. Suppression of repeated values remains in effect until you execute a DBMS SUPREPS with no arguments.

DBMS UPDATE *table_name* SET *col_1* = *expression 1* [, *col_2* = *expression* 2...]

(DBMS UPDATE is not available if server is running SYBASE Ver 3.X)

This command is used to update a row in a browsable table under browse mode. For this command to be invoked:

- The table must be browsable (see DB-Library Reference Manual);
- You must have previously executed DBMS BROWSE

DBMS USE *database_name*

This command changes the current database.

JAM/DB*i*

Release 5

Document Structure

The JAM/DB*i* Manual is printed in one volume. It is divided into the following sections:

- JAM/DB*i* Overview – An overview of the JAM/DB*i* product and the development process. This section describes JAM/DB*i* from the "big picture" viewpoint. It describes all the pieces of a sample JAM/DB*i* application.
- Developer's Guide – A guide to using JAM/DB*i* features. This section is divided into four main sections:—accessing a database with JAM/DB*i* structures, sending JAM values to a database, sending database values to JAM variables, and using transactions.
- Reference Guide – Manual pages for the `dbms` commands, the JAM/DB*i* library functions, and the JAM/DB*i* utilities.
- Notes – A description of features and discussion of topics particular to an engine.
- Appendices – These include lists of keywords, error codes, and suggestions on using JAM more effectively with a JAM/DB*i* application.
- Index

Terminology

Terms will be defined when discussed. However, it is necessary to define a few that will be used throughout the manual.

- *database* — A physical database consisting of tables and other data.
- *vendor* — A supplier of a DBMS engine.
- *engine* — A DBMS product. An engine is identified by a specific vendor and version.

Notation

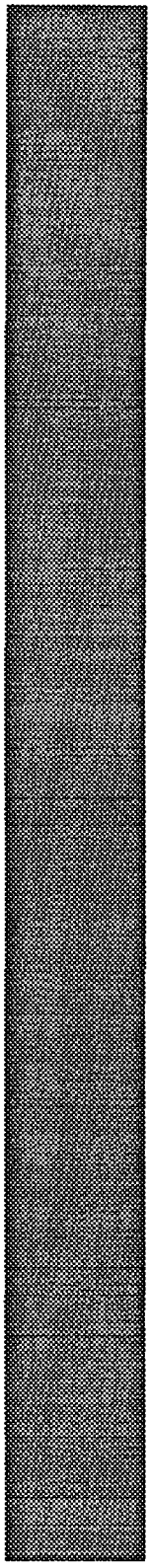
To make this manual easier to use, we use the notation described below.

- `literal` — We use this font for text that you will type verbatim. In particular, we use this font for all examples. We also use it when naming a JAM library function, a JPL command, or a utility.
- `SMALL CAPS` — It is customary to put SQL keywords in uppercase. We follow this convention. In addition, in synopses of `dbms` com-

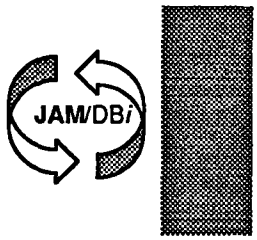
mands, we put dbms keywords in uppercase. Please note that the use of case is purely stylistic. Case is significant only for identifiers—names of fields, columns, tables, variables. functions, etc.

- ***Italics*** — We use bold italics to show where variable or procedure names should appear. Text in this font should be replaced with a specific, appropriate value in an application.
- **[*x*]** — Brackets indicate an optional element. The brackets should not be typed.
- ***x***... — Ellipses indicate that an element may be repeated one or more times.

TABLE OF CONTENTS



JAM/DB*i* Overview



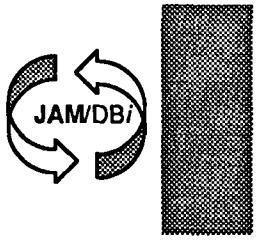
Chapter 1.
Introduction

This document is intended for developers who are using JAM/DB*i*® for the first time, or for those who wish to gain a better understanding of this product. This document is intended to provide a conceptual overview of JAM/DB*i*. It will help you understand and use JAM/DB*i*.

JAM/DB*i* is part of a family of JYACC products. The following table describes the rest of the family:

Product	*Description*
JAM®	JYACC Application Manager
JAM/DB*i* Report writer	Report Writer for JAM/DB*i*
JAM/P*i* for Motif	Presentation interface for the Motif GUI
JAM/P*i* for Microsoft Windows	Presentation interface for Microsoft Windows
JAM/P*i* for Graphics	Presentation interface for Graphics
Jterm®	Color Terminal Emulator optimized for JAM

If you are upgrading from Release 4.8, please read Chapter 21. "Summary of New Features" and Chapter 22. "Release 4.8 Compatibility."



Chapter 2.
What is JAM/DB*i*?

JAM is a software toolkit that aids developers in prototyping and building applications with sophisticated interfaces. **JAM** provides tools for creating screens that accept and display data for end users, and that define the control flow of an application.

JAM/DB*i* is a portable interface between **JAM** applications and relational database systems. It provides facilities for the gamut of data manipulation needs. In particular, a developer may build a JAM/DB*i* application which permits end users to perform any of the following:

- Retrieve values from database tables for display on screens. Queries may be hard-coded, or they be created at runtime according to an end user's specifications.
- Add rows to or delete rows from database tables.
- Update existing rows.
- Create or drop database tables.
- Execute any other function provided by the database's dynamic query interface (e.g., execute a stored procedure, rollback a transaction, etc.).

The rest of this document assumes that you are familiar with **JAM** and the concepts discussed in the **JAM** Overview. In addition, it assumes that you have some database experience.

2.1. COMPONENTS OF JAM/DB*i* ARCHITECTURE

There are several layers in the JAM/DB*i* architecture.

1. JAM Application – This typically includes the following:
   - Menu screens for control flow in the application;
   - Screens for entering new values to a database;
   - Screens for viewing and updating information in a database;
   - Related hook functions.
2. JAM/DB*i* – The interface between a JAM application and a DBMS client library. The interface has a generic part and one or more specific parts called "support routines." A support routine provides access to a particular DBMS product, also called an "engine."
3. DBMS Client Library – The interface that controls all programmed access to a database. This is the interface between JAM/DB*i* and a DBMS. The DBMS controls all access to the database.
4. DBMS Network Services – The network services that connect a user's client library with one or more DBMS servers.
5. DBMS Server.

See the figure below.

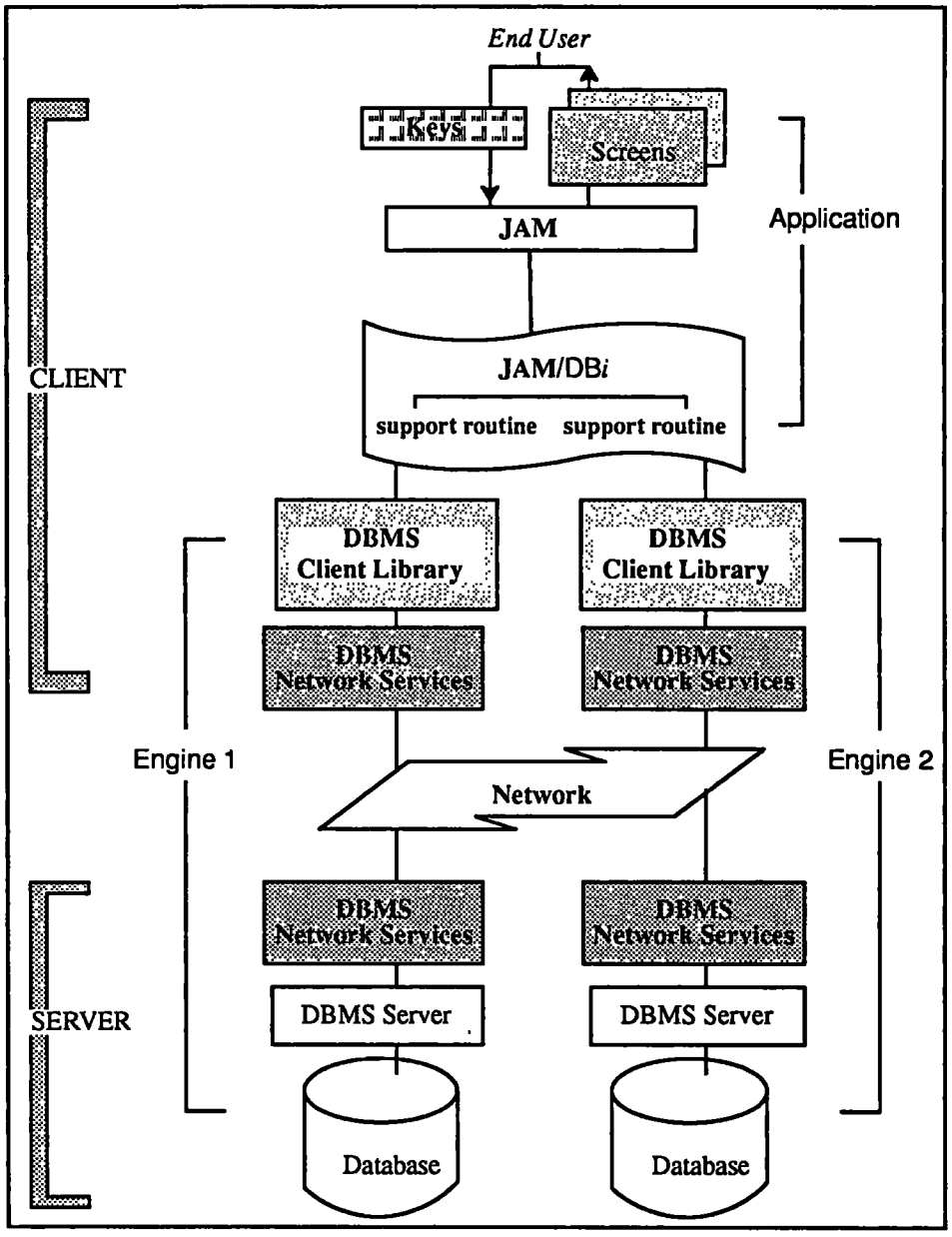


Figure 1: Components of JAM/DB*i* Architecture.

2.2.
COMPONENTS OF JAM/DB*i*

The JAM/DB*i* product is collection of programs and data files. In the sections below, we briefly discuss the main components of the product. For more information, see the README file included with the distribution.

2.2.1.
JAM/DB*i* Libraries

The JAM/DB*i* interface is written using tools provided by your database vendor, either embedded SQL or a C language API (applications programming interface). A JAM/DB*i* developer does not need to write any code using embedded SQL or an API, but in order to link an application he or she must have access the header files and libraries supplied with these tools. The README provided with the JAM/DB*i* distribution names and describes the necessary products.

Each JAM/DB*i* supplies a "common" library and one or more engine-specific libraries. The additional engine-specific libraries are provided so that JAM/DB*i* may support different versions of a database, or support different modes, for example on MSDOS, real mode and Windows mode. The library names are database-specific, usually in the form `lib`***db***`.a` or `llib`***db***`.lib` with ***db*** representing a vendor name. For example, ***db*** may be `ora` for ORACLE or `syb` for SYBASE.The JAM/DB*i* README file names and describes the libraries for your database.

2.2.2.
Source Code

The JAM/DB*i* source code module is `dbiinit.c`. Customized for a particular engine, it specifies header files needed by JAM/DB*i*, declares the name of the support routine for the engine, and sets up some defaults for handling errors and case-sensitivity.

2.2.3.
Header Files

JAM/DB*i* supplies some header files. The file `dmerror.h` defines symbolic constants and integer codes for JAM/DB*i* and DBMS errors. The `README` file provides a complete list of the distribution header files.

2.2.4.
Makefile

Once you have edited the makefile to describe the engine version and the pathname to its installation, you must run the makefile to create the JAM/DB*i* executables, `jamdbi`, `jxdbi`, `f2tbl`, and `tbl2f`. See the installation notes and instructions in the makefile for more information.

2.3. COMPONENTS OF A JAM/DB*i* APPLICATION

New users are sometimes confused about the differences between JAM applications and JAM/DB*i* applications. They share many similarities, as shown in the table below.

JAM Application	JAM/DB*i* Application
JAM Screens	JAM screens
Data Dictionary	Data Dictionary
Hook Functions (JPL and/or C)	Hook Functions (JPL and/or C); Hook functions include database function calls
JAM Executable	JAM/DB*i* Executable

In a JAM/DB*i* application, you can log on, query, or update a database. These functions cannot be performed in a standard JAM application unless you write your own database interface.

If you are familiar with JAM, you are familiar with the two types of JAM executables—the authoring executable and the application executable. (If not, see the introductory chapters of the JAM Programmer's Guide.) Similarly, JAM/DB*i* has two executable versions—the authoring executable, sometimes called `jxdbi`, and the application executable, sometimes

called `jamdbi`. The authoring executable links the developer's hook functions with the JAM Screen Manager, JAM Executive, and authoring libraries, as well as the JAM/DB*i* interface libraries and the engine's libraries. It is used to create and test an application. The application executable, on the other hand, is a runtime program which you may distribute to end users. It does not provide access to the JAM Screen, Keyset, or Data Dictionary Editors.

The tables below list and compare the files which developers must link when creating the executables. We describe the JAM/DB*i* files at the end of the section.

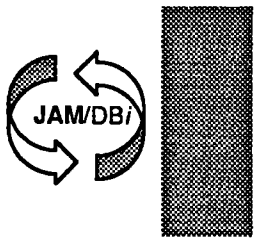
JAM *Authoring* Executable	JAM/DB*i* *Authoring* Executable
`jxmain.o`	`jxmain.o`
`funclist.o`	`funclist.o`
	`dbiinit.o`
JAM Authoring Library (JX)	JAM Authoring Library (JX)
JAM Executive Library (JM)	JAM Executive Library (JM)
JAM Screen Manager Library (SM)	JAM Screen Manager Library (SM)
	JAM/DB*i* Common Interface Library (DM)
	JAM/DB*i* Engine-specific Interface Library (1 or more for each DBMS)
	DBMS Client Library (1 or more for each DBMS)

JAM *Application* Executable	JAM/DB*i* *Application* Executable
`jmain.o`	`jmain.o`
`funclist.o`	`funclist.o`
	`dbiinit.o`
JAM Executive Library (JM)	JAM Executive Library (JM)
JAM Screen Manager Library (SM)	JAM Screen Manager Library (SM)
	JAM/DB*i* Common Interface Library (DM)
	JAM/DB*i* Engine-specific Interface Library (1 or more for each DBMS)
	DBMS Client Library (1 or more for each DBMS)

The JAM/DB*i* Common Interface Library includes the generic routines supported by all engines. It is the interface between JAM and all the engine-specific processing for accessing a database.

The JAM/DB*i* Engine-specific Interface Library is also known as the "support routine." An application must have a support routine for each engine the application uses. The support routine contains all the engine-specific code required by JAM/DB*i*. The JAM/DB*i* Common Interface Library calls an engine's support routine to make the appropriate calls to the DBMS Client Libraries.

The DBMS Client Libraries are supplied by the database vendor. These libraries control all programmed access to a DBMS.



Chapter 3.
JAM/DB*i* *Application Development*

Many of the issues involved in developing a JAM/DB*i* application overlap those involved in developing a JAM application. Here we emphasize issues specific to JAM/DB*i* applications. If necessary, you should see the companion chapter in the JAM Overview for more information on topics like control strings, the Screen Editor, and the Data Dictionary Editor.

3.1.
CREATING AND EDITING APPLICATION SCREENS

Generally, a developer starts creating a new application by creating screens. The developer may use the JAM/DB*i* authoring executable, `jxdbi`, or the JAM authoring executable, `jxform`. In environments where memory is limited, such as MS-DOS, `jxdbi` may be too large and the developer usually must do all development work with `jxform`. If an application screen will be based on a particular table in the database, the developer may use the JAM/DB*i* utility `tbl2f`. This utility creates a JAM screen with a field for each column in the table. JAM assigns field characteristics based on the column's data type. The utility provides a convenient way to develop a maintenance application for a database table, since the utility also creates the JPL procedures for adding, deleting, and updating rows in the table.

3.1.1.

Mapping Columns to JAM Variables

JAM/DB*i* provides a simple way of moving data back and forth between JAM and a DBMS. JAM/DB*i* transfers a SQL statement from the application to the DBMS. When the DBMS returns values, JAM/DB*i* transfers those values to JAM variables.

A JAM variable is any of the following:

- a JPL variable created with a `vars` statement,
- a screen variable
- an LDB entry (i.e., a data dictionary entry with a scope of 2 or greater)

JAM/DB*i* provides two ways of mapping a database column to a JAM variable: automatic mapping and aliasing.

Automatic Mapping

By default, JAM/DB*i* automatically maps a column name in a SELECT statement to a JAM variable with the same name. Suppose the current screen `sales.jam` contains a large scrolling array called `item_no`, and the database table `product` contains a column also called `item_no`. Then,

```
sql SELECT item_no FROM product
```

or,

```
sql SELECT product.item_no FROM product
```

would place the values of column `item_no` in the array `item_no`. Note that a column name always maps to an unqualified field name.

If an application executes

```
sql SELECT * FROM product
```

JAM/DB*i* searches for a JAM variable matching each column in the table `product`. If it finds the variable, it writes the column's values to the variable. If it does not, it ignores the column.

Aliasing

In some circumstances, automatic mapping is undesirable or even impossible. For example, an application may use one screen to show values from two columns with the same (unqual-

ified) name, or a table may have column names that are not valid JAM variable names. In these cases, developers may specify an alias for one or more database columns using the command DBMS ALIAS.[1]

For example, if a table contained a column named `stock^id`, the application could not use automatic mapping because a caret is not a valid character in JAM variable names. The application must set up an alias for the column. For instance,

```
dbms ALIAS "stock^id" stock_id, "company^name" company
sql SELECT stock^id, company^name, dividend FROM stocks
```

JAM/DB*i* would fetch the values of `stock^id` to the alias `stock_id`. It would fetch the values of `company^name` to the alias `company`. (The quotes are used to help JAM/DB*i* parse the column name.) Since no alias was given for the column `price`, JAM/DB*i* would use automatic mapping for this column.

In a DBMS ALIAS statement, a comma separates one column-variable pair from another.

3.1.2.
Data Validation

JAM provides extensive character edits and field validation. In JAM/DB*i* applications, developers use these features to help end users enter and retrieve data easily. Rather than replacing database rules, these edits supply an additional layer of software between the end user and the DBMS. While the tables' rules will ensure the integrity of entered data, a developer can simplify the end users' task—for example, by creating item-selection screens listing valid data. In addition to providing a better interface, an application that performs some validation at the frontend is also more efficient because it reduces the number of trips to the server.

3.2.
ERROR HANDLING

Error handling is an essential component of any database application. In developing a database application, there is often a need for two different approaches to error handling. Developers require low level error messages during the development cycle, while end users usually require high level error messages at runtime.

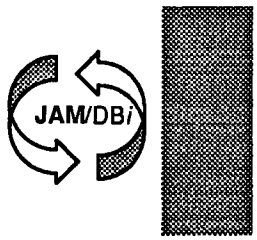
1. Some engines also support aliasing within a SELECT statement. See the section "Using the Engine's SELECT Syntax" on page 82 for more information.

JAM/DB*i* provides several features to assist the developer with these conflicting needs. For any database error, the application has access to a JAM/DB*i* error code and message and an engine error code and message. With the use of a single statement in an application, the developer may alter the way errors are handled and what messages are displayed. It allows the developer to switch easily between running in development mode and prototype mode, and to see the error message appropriate to the mode. The use of two error handlers is not limited to the development cycle. An application may use one error handler for standard endusers and another for the DBA, for instance.

JAM/DB*i* provides several global variables that hold current error and status information. An application does not need to define its own variables to trap this data. The values are accessible from JPL or C.

3.3. ITERATIVE APPLICATION TESTING

Unless your environment has memory constraints, you may use the JAM/DB*i* authoring executable to switch between editing with the Screen and the Data Dictionary Editors, and testing with Application Mode. JAM/DB*i* is turned off in the Screen Editor (draw and test modes) to prevent unintended updates to a database. Without any compilation, you may use Application Mode to test control flow and all JPL procedures in the application. If you are using C hook functions, you must compile and link before testing them.



Chapter 4.
JAM/DB*i* Control Flow

This chapter discusses data flow in JAM/DB*i* applications. To demonstrate the concepts of JAM/DB*i*, it uses a simple example, presenting how the application appears to an end user, and how it appears to a developer. This application is based on the one presented in the JAM Overview. An engine-specific version is supplied in the JAM/DB*i* `samples` directory.

The application consists of three screens. With the first screen, an end user logs on to the database and chooses an area of interest. The next two screens provide access to employee rows stored in three tables. In the application, we use JPL procedures to perform the following:

- Log on and log off a database.
- Query tables, retrieving a single row of values to a JAM screen.
- Query a table, retrieving multiple rows into scrolling arrays.
- Update values in a table.

All the procedures are written in JPL.

This section is not a summary of the product's features. Instead, it uses a fairly simple example to demonstrate control flow in a JAM/DB*i* application. An understanding of the concepts discussed here will help you understand the rest of this document.

Developers interested in creating their own "quick start" application should consider using the utility `tbl2f` to build a small application. `tbl2f` is documented in the *Reference Guide* in this document.

4.1. SAMPLE APPLICATION – USER'S VIEW

The first screen presented to the end user is `mainscrn.jam`.

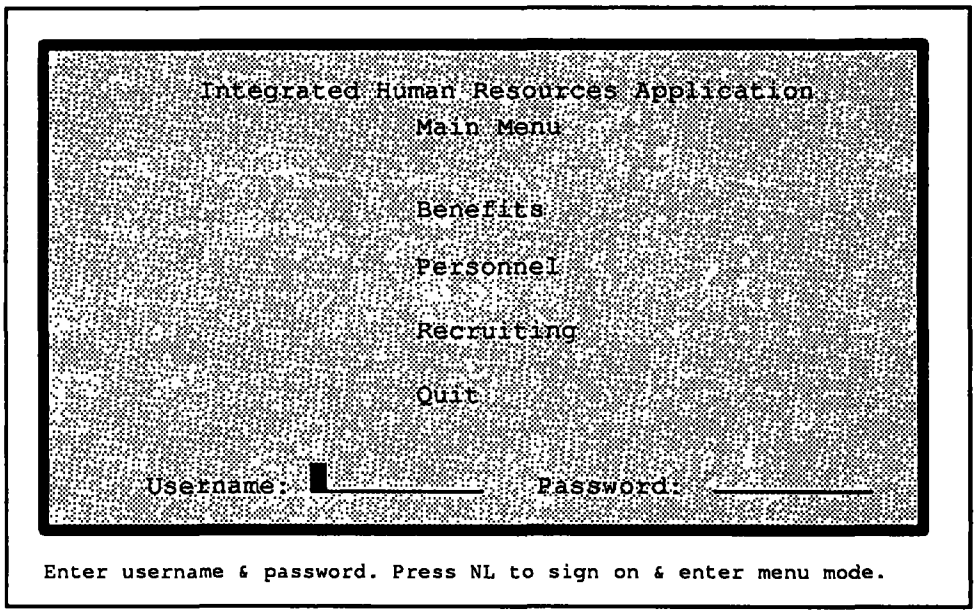


Figure 2: Human Resources Application Main Menu `mainscrn`.

The user must enter a user name and password. If the user has permission to log on, JAM logs on the user and toggles the screen from data entry mode to menu mode. When the user chooses an item from the menu, JAM displays the appropriate screen. If the user chooses `Personnel`, JAM displays the screen `empscrn` shown below.

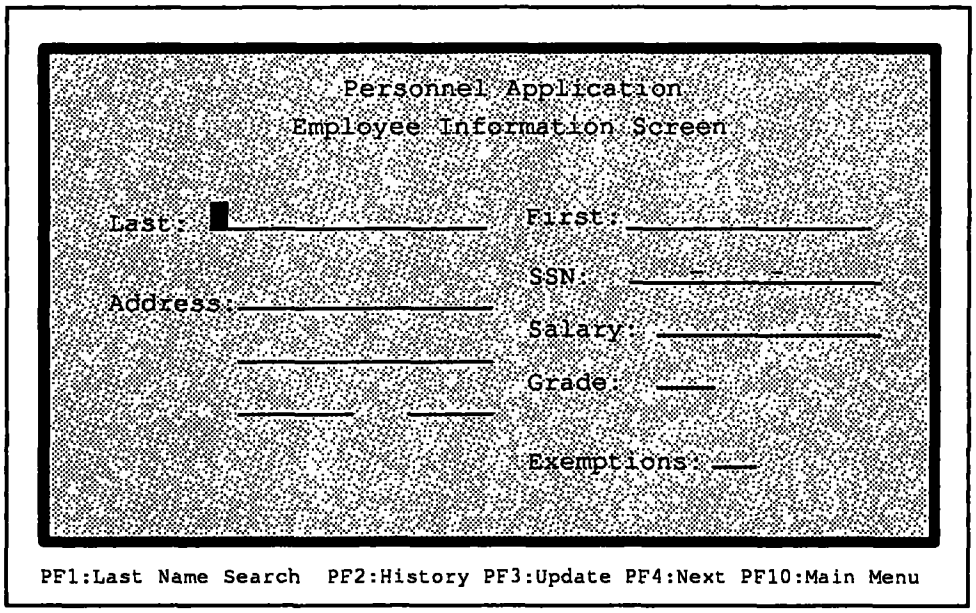


Figure 3: Personnel Application Employee Screen `empscrn`.

The end user enters data in the screen `empscrn.jam` to query the database, and to update rows. The user queries the database by typing an employee surname in the first field and pressing `PF1`. If more than one employee has the same last name, the rows will be retrieved one at a time. The user may press `PF4` to see the next employee row with the specified last name. If the user presses `PF1` without supplying a name, the application retrieves all employee rows in alphabetical order.

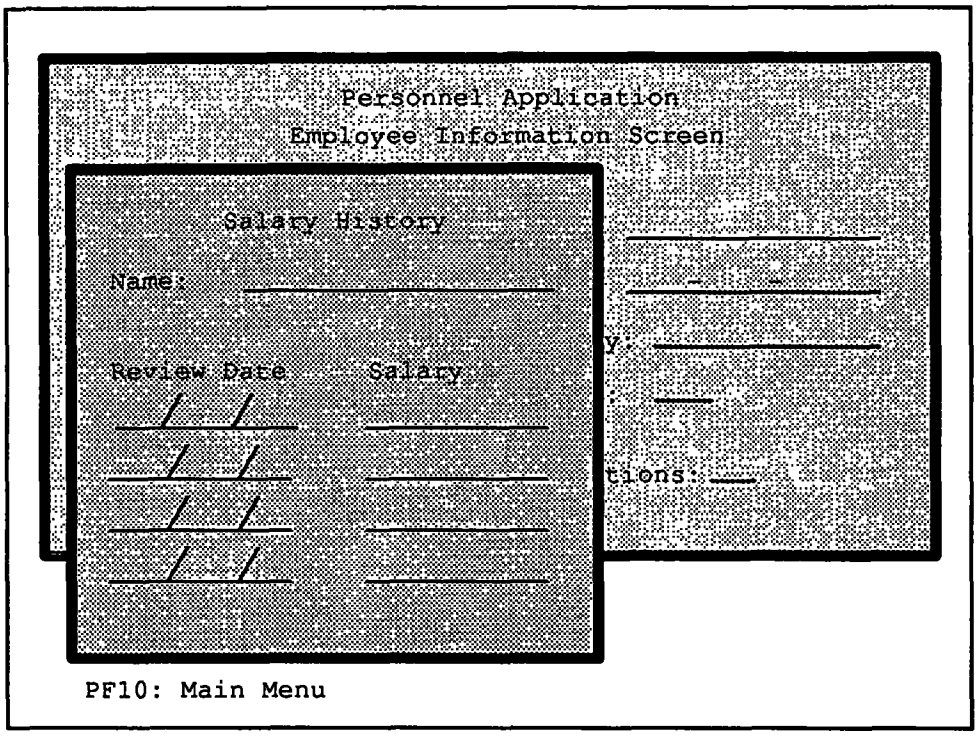


Figure 4: Personnel Application Salary History Window `salhist`.

When JAM displays a row, the user may press the `PF2` key to review the employee's salary history.

4.2.
SAMPLE APPLICATION – DEVELOPER'S VIEW

In this section, we show how the main components of this application appear to a developer. In particular, we describe the database tables, JAM screens, and JPL functions constituting the application.

4.2.1.

Database Tables emp, acc, and review

Below are sample SQL statements for the application tables. Please note that some engines use different names for column datatypes. The table entries represent seven employees.

The table emp has eight columns. Each row stores an employee's social security number, name, home address, and current grade.

```
CREATE TABLE emp (
   ssn       CHAR(11)        NOT NULL,
   last      CHAR(20),
   first     CHAR(12),
   street    CHAR(20),
   city      CHAR(15),
   st        CHAR(2),
   zip       CHAR(5),
   grade     CHAR(1))
```

ssn	last	first	street	city	st	zip	grade
038-68-6826	Jones	Barnabus	321 West 11 St	Albuquerque	NM	87124	C
122-98-6541	Aumond	Hilary	11-12 Front St	Albuquerque	NM	87124	E
122-99-4102	Jones	Michael	5 Maple Drive	Albuquerque	NM	87124	B
139-42-1651	Blake	Norman	34 Concord Ave	Albuquerque	NM	87124	D
154-32-6610	Cory	Richard	411 Ann St	Albuquerque	NM	87124	D
310-77-3997	Grundy	Janet	70-2 Poe Ave	Albuquerque	NM	87124	D
310-32-0084	Jones	John P	9 Vern Terrace	Albuquerque	NM	87124	D

Figure 5: Table emp

Table acc has three columns. Each row stores an employee's social security number, current salary, and a number of tax exemptions.

```
CREATE TABLE acc (
   ssn       CHAR(11)        NOT NULL,
   sal       NUMERIC(10.2),
   exmp      NUMERIC(1))
```

ssn	sal	exmp
038–68–6826	29500.00	1
122–98–6541	37800.00	3
122–99–4102	26000.00	3
139–42–1651	89500.00	2
154–32–6610	43100.00	4
310–77–3997	38000.00	1
310–32–0084	47500.00	5

Figure 6: Table `acc`

Table `review` has four columns. Each row stores an employee's social security number, a hire date or review date, a new salary if it has changed since the previous review, and a new grade if it has changed since the previous review. If `newsal` or `newgrade` is null, the employee was reviewed but there was no change in salary or grade.

```
CREATE TABLE review (
   ssn       CHAR(11)          NOT NULL,
   revdate   DATE              NOT NULL,
   newsal    NUMERIC(10.2),
   newgrade  CHAR(1))
```

ssn	revdate	newsal	newgrade
038–68–6826	12/13/90	49500.00	C
038–68–6826	12/11/89	45000.00	NULL
038–68–6826	12/15/88	NULL	NULL
038–68–6826	12/14/87	38500.00	D
122–98–6541	04/10/90	37800.00	NULL
122–98–6541	04/08/89	31000.00	E
122–99–4102	05/01/90	29000.00	NULL
122–99–4102	05/01/89	25200.00	E
139–42–1651	11/12/90	89500.00	NULL
139–42–1651	11/08/89	81000.00	B
139–42–1651	11/10/88	67500.00	C
139–42–1651	11/10/87	NULL	NULL
139–42–1651	11/08/86	53000.00	D
154–32–6610	02/01/91	43100.00	D
310–77–3997	07/16/90	38000.00	D
310–77–3997	07/14/89	30000.00	E

310–32–0084	03/01/91	47500.00	D
310–32–0084	03/01/90	43000.00	E

Figure 7: Table `review`

The sample application permits an enduser to view rows from these tables and to update data in some columns.

4.2.2.
Source Module `dbiinit.c`

To save memory, JAM supplies several features as optional subsystems. These subsystems include soft keys and alternate scrolling as well as DB*i*. The JAM/DB*i* subsystem is installed by setting the `DBI` macro in `jmain.c` (or `jxmain.c`) or by setting a compiler directive.

The application must initialize an engine with the function `dm_init` before making a connection. Developers may call this function directly or they may use the vendor structure in `dbiinit.c` to store the engine initialization information. JAM/DB*i* supplies a version of this file customized for your engine.

An excerpt from `dbiinit.c` is shown below. The boldface text shows the statements that would install a fictional DBMS called XYZdb for the sample application.

```
#include "smdefs.h"
#include "dmerror.h"
#include "smusrdbi.h"
#include "dmuproto.h"

#if DBIVENDORLIST
/* Support routine function prototypes */
/* Copy the following line for each support routine */
/* that is to be installed.  Uncomment each copy,   */
/* and replacle 'support_routine' with the name of  */
/* the support routine to be installed.             */

/* extern int support_routine PROTO((int)); */
extern int dm_xyzsup PROTO((int));

/* Add one entry to the following structure for each database support*/
/* routine that is to be installed.  The form of each entry is as    */
/* follows:                                                          */
/*                                                                   */
```

```
/*      { "engine_name", support_routine, case_flag, (char *) 0 },    */
/*                                                                    */
/* Replace 'engine_name' with the name of the database you are        */
/* installing.  Replace 'support_routine' with the name of the        */
/* support routine for that database.  Replace 'case_flag' with       */
/* one of:                                                            */
/*      DM_DEFAULT_CASE          (Use the default value for the       */
/*                                case_flag specified in              */
/*                                the support routine)                */
/*      DM_PRESERVE_CASE         (No case conversion is performed on */
/*                                database columns)                   */
/*      DM_FORCE_TO_LOWER_CASE   (Maps upper and mixed case column    */
/*                                names to lower case jam field       */
/*                                names during a database query)      */
/*      DM_FORCE_TO_UPPER_CASE   (Maps lower and mixed case column    */
/*                                names to upper case jam field       */
/*                                names during a database query)      */
/*                                                                    */
/* The last member in the structure is for future expansion.
*/static vendor_t vendor_list[] =
{
/*      { "engine_name", support_routine, case_flag, (char *) 0 },    */
        { "xyzdb", dm_xyzsup, DM_FORCE_TO_LOWER_CASE, (char *) 0 },
        { (char *) 0, (int (*)()) 0, (int) 0, (char *) 0 }
};
```

The entry

```
{ "xyzdb", dm_xyzsup, DM_FORCE_TO_LOWER_CASE, (char *) 0 }
```

contains four elements. The first,

```
"xyzdb"
```

names the engine for the application. It may be any name the developer wishes; an abbreviated vendor name is common. The second element,

```
dm_xyzsup
```

names the engine's support routine. This support routine is supplied in a library as a part of the JAM/DB*i* distribution and its name is documented in the README file. The third,

```
DM_FORCE_TO_LOWER_CASE
```

tells JAM/DB*i* how to handle the case of column names when executing a SELECT. This flag tells JAM/DB*i* to convert column names to lower case when searching for JAM variable destinations for a SELECT. Therefore, the application uses lower case for screen, LDB, and JPL variables that are targets for database columns.

Any developer-written C hook functions are installed in `funclist.c`. Since the sample application uses only JPL it uses the distributed `funclist.c` without any modifications. For more information on `funclist.c` or prototyped functions, see the JAM *Programmer's Guide.*

4.2.3.

Data Dictionary and Initialization File

The application's data dictionary has three types of entries. They are the following:

- constants named and initialized for JAM/DB*i* errors
- variables for passing database values between screens at runtime

See the figures below.

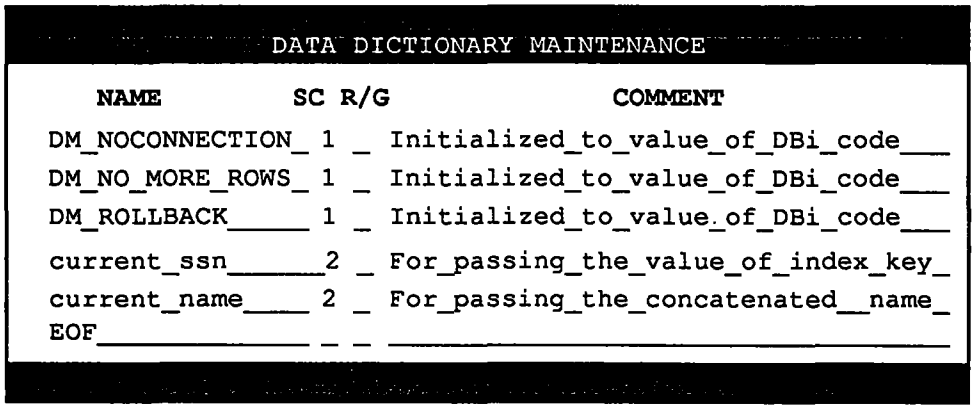


Figure 8: Developer's View of the Data Dictionary.

```
# const.ini
# This file inializes LDB constants.
# Values correspond to those in DBi header file dmerror.h

"DM_NO_MORE_ROWS"     "53256"
"DM_ROLLBACK"         "53263"
"DM_NOCONNECTION"     "53271"
```

Figure 9: Developer's View of the Constants' Initialization File.

The DM_ variables are named after symbolic constants in the JAM/DBi file dmerror.h. Note that the scope of these variables is 1. At runtime, these values are treated as constants by the local data block (LDB) initialization. A constants' initialization file, such as const.ini, assigns the values to the constants. See Appendix B. for the complete list of JAM/DBi error codes.

The entries current_ssn and current_name are used to pass database values between screens at runtime.

4.2.4.

JAM Screens

There are three application screens.

Each of the screens uses one or more JPL modules. There are several ways of storing and accessing JPL procedures and modules. A module is one or more JPL procedures. The type of module describes how it is stored—in a file, as a miscellaneous field edit, etc. See the JPL Guide in the JAM manual for a discussion of these topics.

Main Screen

The screen mainscrn.jam contains a menu and two data entry fields, uname and pword. The screen opens in data entry mode. The field pword has a procedure in its JPL field module. When the end user tabs from this field, the procedure installs an error handler and attempts to log the end user onto the database with the user name and password entered in the fields. If log on is successful, it calls the built-in function jm_mnutogl to toggle the screen from data entry mode to menu mode.

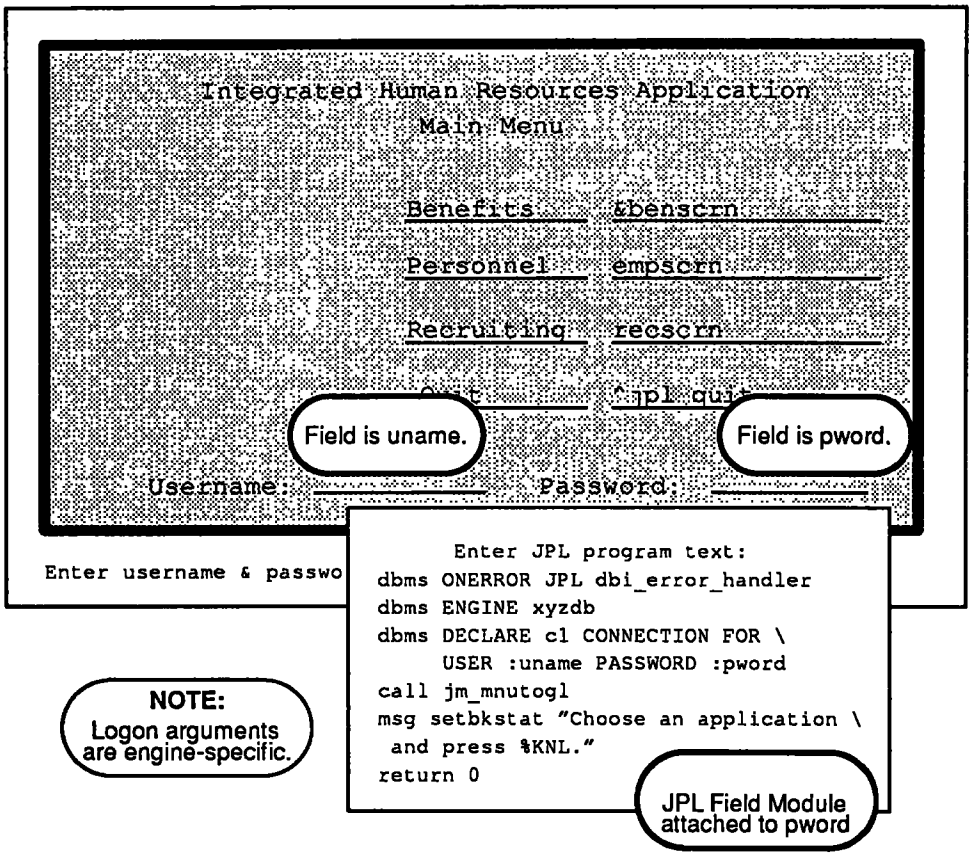


Figure 10: Human Resources Application Main Menu, `mainscrn`.

JPL Field Module Attached to Field `pword`

The first statement of the procedure sets up error processing for the rest of the application. The DBMS ONERROR statement installs the JPL procedure `dbi_error_handler` as the application's error handler. Whenever a JAM/DB*i* error occurs, JAM/DB*i* passes three arguments to the procedure—the text of the statement that failed, the name of the current engine, and an error flag—and executes the procedure. A sample error handler is shown in Figure 11.

The statement DBMS ENGINE names `xyzdb` as the default engine. Since only one engine was installed in `dbiinit.c`, this statement is optional.

The statement DBMS DECLARE CONNECTION attempts to log the user on to a database server. If log on is successful JPL continues executing the procedure: it toggles `mainscrn` from data entry mode to menu mode and displays a new status line message.

JPL Procedure for Error Handling, `dbi_error_handler`

If the log on is unsuccessful, JAM/DBi immediately calls the installed error handler `dbi_error_handler`:

```
proc dbi_error_handler
parms stmt code flag
# All DM_ variables are constants in the LDB.

# If stmt failed because the user did not logon, prompt user to return
# to main screen.

    if (@dmretcode == DM_NOCONNECTION)
    {
        msg emsg "Not logged on. Press %KPF10 to restart."
    }
    else
    {

# For all other errors, display the JAM/DBi message and any database
# error message.

        msg emsg @dmretmsg
        if @dmengerrmsg != ''
            msg emsg @dmengerrmsg
    }

# For all errors, return the abort code (1) to abort the JPL procedure
# where the error occurred. If 0 were returned, the procedure where the
# error occurred would continue executing.
    return 1
```

Figure 11: Sample JPL Error Handler for Human Resources Application.

Note that three arguments are automatically passed to any error handler installed with DBMS ONERROR:

- the text of the statement that failed
- the name of the engine in use when the error occurred
- a flag indicating that this procedure was called because an error occurred

After receiving the arguments, the procedure examines the error code. Note the use of the variables `@dmretcode`, `@dmretmsg`, and `@dmengerrmsg`. These are global variables defined and maintained by JAM/DB*i*. If there is an error executing a `sql` or `dbms` statement, JAM/DB*i* writes a JAM/DB*i* error code to `@dmretcode`, a JAM/DB*i* error message to `@dmretmsg`, an engine-specific error code to `@dmengerrcode` and an engine-specific error message to `@dmengerrmsg`. The application may use these variables in JPL statements such as `if` or `msg` when processing for errors.

The procedure first checks if the user is connected to an engine. For instance, if the user has a mouse and clicks on a menu choice, he or she may move to the next screen before logging on. However, once he or she attempts to view employee data, JAM/DB*i* will return an error because there is no connection to the database. In case of this error, the error handler prompts the user on how to recover—pressing PF10 returns the user to the top-level form where a user name and password may be entered.[2] Recall that `DM_NOCONNECTION` was defined as an LDB constant (Figure 8 and Figure 9).

For all other errors, the error handler displays a standard JAM/DB*i* error message, and also an engine-specific message if there is one in the global JAM variable `@dmengerrmsg`. For example, the user may enter a user name and password on `mainscrn`, but the logon may fail for some reason. In such a case, the handler first displays a JAM/DB*i* message telling the user that the operation failed. Next it displays the engine-specific message further describing the failure—for example invalid user name, password is required, or the server is not available, etc.

In addition to displaying messages, the error handler also determines whether to continue or abort execution of the JPL procedure where the error occurred. If the error handler returns 0, JPL continues execution at the next statement after the one that failed. If the handler returns 1, JPL aborts the procedure and returns control to the procedure's caller.

The sample error handler returns the abort code (1) for all errors. Therefore, if logon fails, JPL does not execute the rest of the procedures in the JPL field module of `pword`. Therefore, it does not execute the statements which toggle the screen to menu mode and change the status line message. Instead, it returns control to the procedure's caller, in this case JAM.

There are many advantages to JAM/DB*i*'s error handling features. Most notably, it gives developers both generic and vendor-specific means of handling errors. In addition, the error handler like the rest of the application is easily prototyped. In early stages of the application, the error handler may simply display all error messages. As the application grows, the developer may enhance the error handler, adding special processing and messages for particular errors. The error handler may also be written in C.

2. Of course, a target list on the menu control strings on `mainscrn` could prevent this. Each menu choice could call a procedure that verifies that the has user logged on before opening the next form or window. See the *Author's Guide* in the JAM documentation for information on using target lists.

To use a JPL error procedure most efficiently, the procedure should be in a public module. See the *JPL Guide* for details.

Menu Choices on `mainscrn`

Once in menu mode on `mainscrn`, the user may choose among the three applications—`Benefits`, `Personnel`, `Recruiting`—or may quit.

The last option on the menu, `QUIT`, calls the JPL procedure `quit` to log the user off the database and exit the application. Logoff may be executed with the statement,

```
dbms CLOSE_ALL_CONNECTIONS
```

The rest of this chapter describes the `Personnel` option.

Employee Screen

If the user chooses `Personnel` from the menu, JAM opens the form `empscrn` shown below.

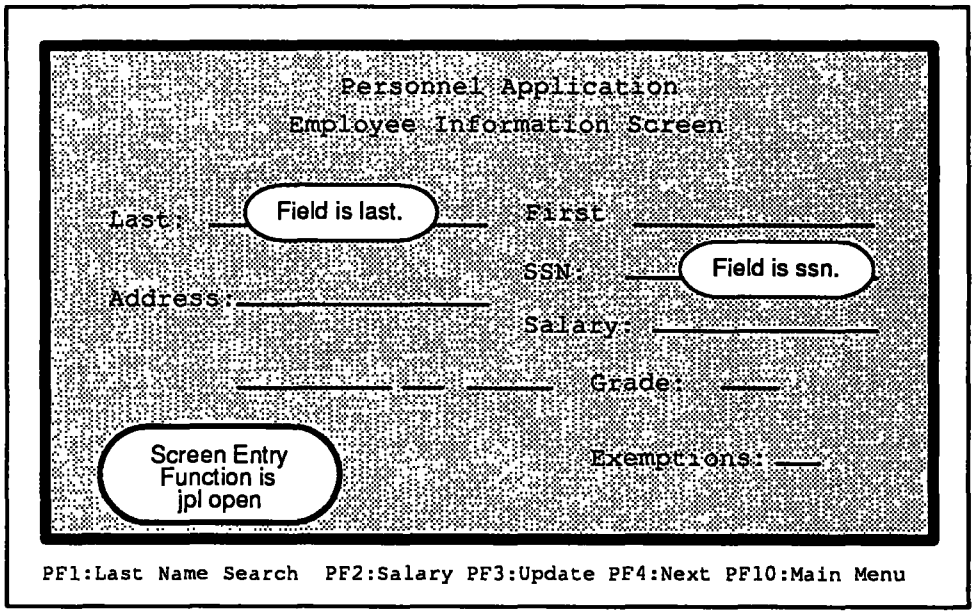


Figure 12: Personnel Application Employee Screen `empscrn`. The social security, salary, and grade fields are protected from data entry.

The screen empscrn.jam is used to update and display data from the database. The screen has eleven fields: last, first, street, city, st, zip, ssn, grade, sal and exmp. The function keys PF1 and PF2 are associated with JPL functions that query the tables acc, emp, and review. The PF3 key permits a user to update name and address values in the table emp, and the number of exemptions in the table acc. If the end user wishes to scroll through the employee records, pressing the PF4 will fetch a new row. The PF10 key returns the user to the menu screen.

The fields ssn, sal, and grade are protected from data entry. The end user may update an employee's name, address, or number of exemptions. The application assumes that an employee's social security number should not change. An employee's salary and grade may only be changed after an employee review. We assume that such information is entered in another application. Developers, of course, could write a function that permits certain users to change data in protected fields. The JAM Programmer's Guide documents the library functions necessary for this type of processing.

Below is the text of the JPL procedures for empscrn and an explanation of the procedures.

JPL Procedure open

```
proc open
msg setbkstat "\
%KPF1 Last Name Search %KPF2 Salary  %KPF3 Update  %KPF4 Next \
%KPF10 Main Menu"

dbms DECLARE emp_cursor CURSOR FOR \
SELECT emp.first, emp.last, emp.street, emp.city, emp.st, emp.zip,\
emp.ssn, emp.grade, acc.sal, acc.exmp FROM emp, acc \
  WHERE emp.ssn=acc.ssn AND emp.last LIKE ::parm_last \
  ORDER BY emp.last, emp.first
return 0
```

Figure 13 a : JPL screen module for empscrn.

This procedure is the screen entry function. The msg statement displays a status line message which describes the screen's control keys. The second statement declares a cursor, emp_cursor, for a SELECT statement. The SELECT is just like a SELECT statement executed in a DBMS interface, except for the argument ::parm_last. This argument is a *binding parameter*. JAM/DB*i* will not know its value until the end user presses the PF1 key which executes the cursor. Executing the cursor will execute the SELECT and fetch data to the screen.

JPL Procedures `search` and `next`

```
proc search
if last == ""
  dbms WITH CURSOR emp_cursor EXECUTE USING parm_last = '%'
else
  dbms WITH CURSOR emp_cursor EXECUTE USING parm_last = last
if @dmretcode == DM_NO_MORE_ROWS
  msg emsg "There are no employees with the surname :last ."
return 0

proc next
dbms WITH CURSOR emp_cursor CONTINUE
if dbi_retcode == DM_NO_MORE_ROWS
  msg emsg "There are no more rows."
return  0
```

Figure 13 b: Continuation of JPL screen module for `empscrn`. These functions are executed with PF1 and PF4.

The procedure `search` begins by checking if the field `last` is empty. If it is empty, the procedure executes `emp_cursor` (declared in Figure 13 a) using the wild character '%'. Thus, if the end user presses `PF1` without supplying a surname, JAM/DB*i* fetches all the employee rows one at a time in alphabetical order.

If the field `last` is not empty, the procedure executes `emp_cursor` with the surname entered in the field. If two or more employees have the same surname, more than one row is returned. The enduser presses the `Next` key to see the next available record.

For example, if the end user entered the surname "Jones" in the field named `last`, the DBMS would find three qualifying employees in the database. JAM/DB*i* displays the information on employee Barnabus Jones when the `PF1` key is pressed. When the `PF4` key is pressed, JAM/DB*i* displays the next employee in the `SELECT` set, John P. Jones. When the `PF4` is pressed a second time, JAM/DB*i* displays the information on the final employee, Michael Jones. If the user presses the `PF4` key a third time, the procedure tells the user that there are no more rows in the `SELECT` set.

The procedure can tell the user when all rows have been displayed because the engine sends a no-more-rows signal if the application tries to fetch more rows than there are in the `SELECT` set. When this signal is returned, JAM/DB*i* writes the value of the `DM_NO_MORE_ROWS` code to the global variable `@dmretcode`. The JPL procedure knows the value of `DM_NO_MORE_ROWS` because a variable of the same name was defined as an LDB constant (Figure 8) and was assigned a value by the initialization file `const.ini` (Figure 9).

JPL Procedure check_ssn

```
proc check_ssn
 if ssn != ""
   return 0
 msg emsg "\
 A social security number is required. Please enter an employee's\
 last name and press %KPF4 to retrieve the necessary information.\
 When a record is displayed, press %KPF2 to see the salary history\
 or press %KPF3 to make an update."
return 1
```

Figure 13 c: Continuation of JPL screen module for empscrn.

The procedure check_ssn is used by the procedures salhist and update. It verifies that the user has entered a social security number. If no number is given, check_ssn displays an error message.

JPL Procedure salhist

```
proc salhist
  vars jpl_retcode
  retvar jpl_retcode

  jpl check_ssn
  if jpl_retcode == 0
  {
    cat current_ssn ssn
    cat current_name first " " last
    call jm_keys PF14
  }
  return 0
```

Figure 13 d: Continuation of JPL screen module for empscrn.This function is executed with PF2.

The end user presses the PF2 key to review an employee's salary history. The procedure begins by setting up a return variable and calling the procedure check_ssn. The procedure check_ssn (Figure 13 c) tests whether the field ssn is empty. If ssn is empty, the procedure displays a message telling the user to press the PF1 key before requesting a history. The return code from check_ssn determines whether salhist continues executing. If the code is 0 (i.e., ssn is not empty) the procedure continues.

This routine copies the current employee social security number to the LDB variable current_ssn, and concatenates the values of first and last in the LDB variable current_name. The values are copied to the LDB so that the salary history screen may use them.

The statement `call jm_keys` executes a control string. The JAM control string window for `empscrn` contains the entry

```
PF14   &(9,25)salhist
```

which opens the screen `salhist` at row 9, column 25. The discussion of the `salhist` screen begins on page 33.

JPL Procedure `update` and Related Procedures

```
proc update
vars jpl_retcode ans
retvar jpl_retcode
 jpl check_ssn
 if jpl_retcode == 0
 {
    msg query "Update this record now?" ans
    if ans
      jpl tran_handle upd_emp
  }
  return 0

proc tran_handle
parms subroutine
vars tran_error
retvar tran_error
  jpl :subroutine
  if tran_error
  {
     msg emsg "Rolling back transaction."
     dbms ROLLBACK
  }
  else
     msg emsg "Transaction successful."
  return 0

proc upd_emp
  dbms BEGIN
  sql UPDATE emp SET last=:+last, first=:+first, \
    street=:+street, city=:+city, st=:+st, zip=:+zip WHERE ssn=:+ssn
  sql UPDATE acc SET exmp=:+exmp WHERE acc.ssn=:+ssn
  dbms COMMIT
  return 0
```

NOTE:
Transaction commands are engine-specific.

Figure 13 e: End of JPL screen module for `empscrn`. The procedure `update` is executed with PF4.

The procedure `update` begins by setting up a return variable and calling the procedure `check_ssn`. The procedure `check_ssn` (Figure 13 c) tests whether the field `ssn` is empty. If `ssn` is empty, the procedure displays a message telling the user to press the `PF1` key before performing an update. The return code from `check_ssn` determines whether `update` continues executing. If the code is 0 (i.e., `ssn` is not empty) the procedure continues, asking the user to confirm the update. If the end user enters the value of `SM_YES` (typically "y"), the procedure passes the name of a subroutine `upd_emp` to a transaction handler `tran_handle`.

The procedure `tran_handle` is a generic procedure that may be used to execute any transaction. It receives one argument, the name of a subroutine that contains the transaction statements. Before calling the subroutine, however, `tran_handle` defines and declares a return variable `tran_error`. After calling the subroutine, `tran_handle` checks if `tran_error` is non-zero; a non-zero value signals that an error has occurred and that `tran_handle` must execute a rollback. This method permits the application to test and rollback for both JAM and JAM/DB*i* errors. The return code for a JAM error is always –1, and the return code from the sample error handler `dbi_error_handler` is 1.

The procedure `upd_emp` is engine-specific. Some engines, such as ORACLE, begin a transaction with the command `DBMS AUTOCOMMIT OFF`. If you are building this application, please consult the engine-specific documentation.

Note the use of `:+`***variable*** in the `UPDATE` statements. This is the colon-plus preprocessor. Before executing the statement, JPL replaces each instance of `:+`***variable*** with the value of ***variable*** in a format suitable for the engine.

For example, if the screen contained the following values,

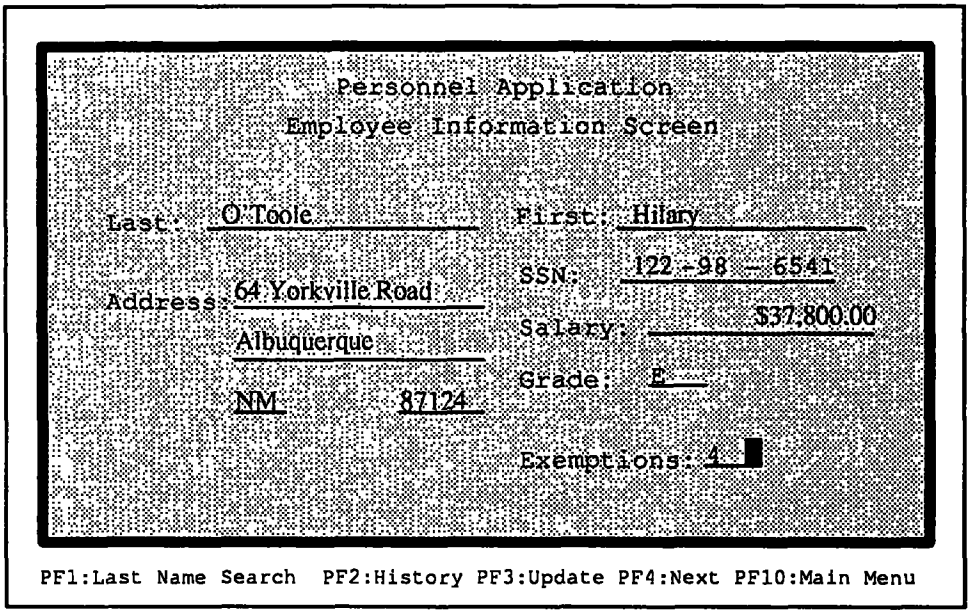


Figure 14: Screen Editor Entry Screen

and assuming that the fields `last`, `first`, `street`, `city`, `st`, and `zip` are all character fields with no special edits, and `exmp` is a `digits only` field, the procedure would execute something like the following,

```
UPDATE emp SET last='O''Toole', first='Hilary', \
   street='64 Yorkville Road', city='Albuquerque',\
   st='NM', zip='87124' WHERE ssn='122-98-6541'

UPDATE acc SET exmp=4 WHERE acc.ssn='122-98-6541'
```

Note that the colon-plus processor formats character data differently than numeric data. Character strings are automatically enclosed in quotes and embedded quotes in character strings are escaped. Numeric values are not quoted. This formatting is engine-specific and is handled automatically by JAM/DB*i*. This topic is covered in detail in the *Developer's Guide* of this manual.

Salary History Screen

If the user presses the `Salary History` key while an employee row is displayed, JAM opens the window `salhist`, shown below.

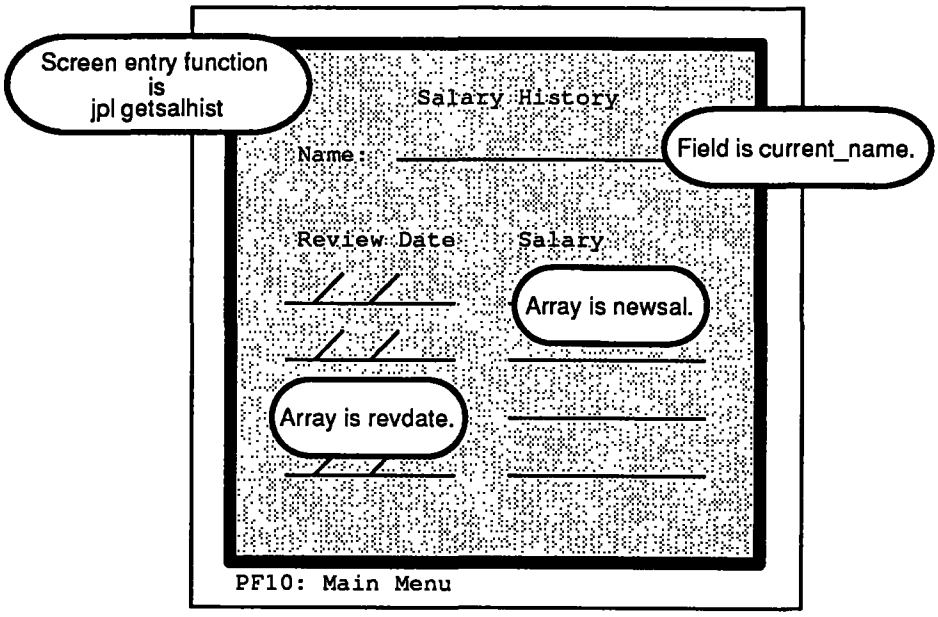


Figure 15: Developer's View of `salhist`.

Upon opening `salhist`, JAM calls the JPL function `getsalhist`, shown below.

```
proc getsalhist
  msg setbkstat "    %KPF10 Main Menu"
  sql SELECT revdate, newsal FROM review WHERE ssn=:+current_ssn
return
```

Figure 16: Developer's View of the JPL Screen Module for `salhist`.

Remember that `current_name` and `current_ssn` are LDB variables (Figure 8). The procedure `salhist` on the previous screen concatenated the values of `first` and `last` in the variable `current_name`, and copied the social security number from `ssn` to `current_ssn` (Figure 13 d). The field `name` is protected from data entry and tabbing.

If `empscrn` is displaying the data belonging to the employee Barnabus Jones when the `History` key is pressed, then `getsalhist` executes

```
SELECT revdate, newsal FROM review \
   WHERE ssn='038-68-6826'
```

and JAM displays the following data:

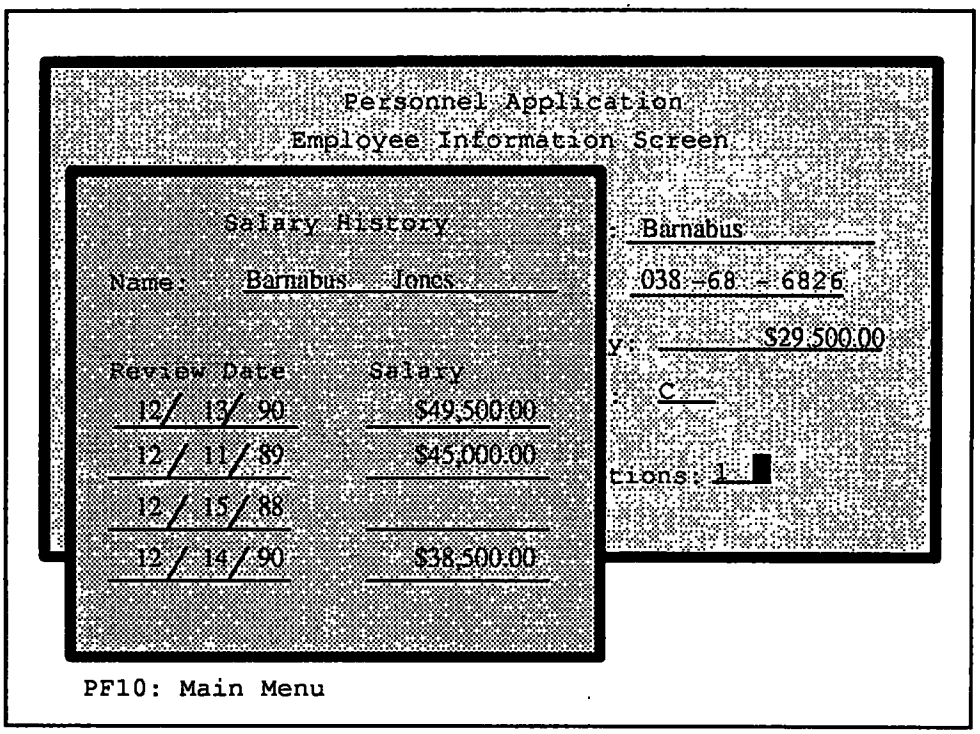


Figure 17: Personnel Application Salary History Window `salhist`.

The arrays `revdate` and `newsal` are large scrolling arrays. The user may press the page-up and page-down keys (JAM logical keys `SPGU` and `SPGD`) to view all the rows. The user may press the `EXIT` key to return to `empscrn`, or press the `Main Menu` key to return to the application's first screen.

4.3.
JAM/DB*i* CONTROL FLOW SUMMARY

In this section we review control flow between JAM and a database, using the Personnel Application as an example.

In JAM/DB*i* applications, database queries are embedded in hook functions written in JPL or C. Hook functions are explained in detail in the JAM Programmer's Guide. Here we note that the choice of hook function and the choice of coding language affects the construction and the control flow of a query.

4.3.1.
Variable Substitution

Applications usually require that the end user specify search criteria at runtime. In these cases, an end user enters data into screen fields and JAM uses the fields' contents in the SELECT statement. JAM provides several ways of accessing field contents at runtime. They are the following:

- colon preprocessor
- `sm_getfield` and related functions
- argument of a field function

The colon preprocessor is an easy and efficient method of accessing field contents at runtime. JAM invokes the colon preprocessor on the arguments of a control string beginning with a caret. Therefore, developers may pass the contents of JAM variables as parameters to the control function. If the control string is passing more than one parameter to a C function, the function should be installed as a prototyped function. See the Author's Guide for more information on colon preprocessing and control strings. See the Programmer's Guide for information on prototyped and control string functions.

JAM invokes the colon preprocessor each time it executes a JPL statement. Therefore, JPL developers may access field and LDB values within a JPL procedure. (See the JPL Guide for information on colon preprocessing with JPL commands.)

JAM also invokes the colon preprocessor on the arguments of the JAM/DB*i* library functions `dm_sql` and `dm_dbms`. In addition, C developers may use the library function `sm_getfield`, or a host of variants, to access runtime values. See the Programmer's Guide for descriptions of these JAM functions.

In JAM/DB*i* applications, colon preprocessing is usually preferable to the functions like `sm_getfield` because it automatically formats data in an engine's style.

4.3.2.
Cursors

SQL vendors support cursors as a part of the interface to custom applications such as `jamdbi`. A cursor is a SQL object that allows an application

- to fetch rows from a SELECT set incrementally
- to use more than one SELECT set at a time

- to improve efficiency when executing a SQL statement many times

On each connection, JAM/DB*i* automatically creates a cursor for SELECT statements. For some engines, it also creates another cursor non-SELECT statements. These cursors are known as the "default" cursors. The JPL command `sql` and the library function `dm_sql` always use a default cursor.

In addition, developers may declare cursors with the command DBMS DECLARE CURSOR. A declared cursor is always named and associated with a SQL statement. Named cursors are executed with the JPL command `dbms` or with the library function `dm_dbms`. In JPL, the statement is

```
dbms WITH CURSOR cursor EXECUTE
```

Executing a named cursor executes the statement that was associated with the cursor at its declaration.

Fetching a SELECT Set Incrementally

When creating screens for displaying database values, the developer may, at best, only approximate the number of rows which will be in a SELECT set fetched by the application. Therefore, JAM/DB*i* needs a mechanism for handling SELECT sets that contain more rows than can be held by the JAM destination variables at one time. If, for example, a SELECT set contains 100 rows, but destination variables have only twenty occurrences each, JAM/DB*i* cannot fetch more than 20 rows at a time. Therefore, it needs a "place holder" in the set so that after fetching rows 1 through 20 when the SELECT is executed, it can fetch rows 21 through 40 when DBMS CONTINUE is first executed, rows 41 through 60 when DBMS CONTINUE is executed a second time, and so on. A cursor acts as such a placeholder.

Using Multiple SELECT Sets

JAM/DB*i* automatically creates one default cursor for SELECT statements. Very often, however, applications use two or more SELECT sets concurrently. This would permit a user, for example, to select many item "summary" rows where he or she may position the screen cursor and then execute one or more SELECTS for "detail" rows further describing the item. After viewing detail rows, the user may contain viewing the item summary rows.

This was the approach in the sample application where we used a named cursor to select employee rows and the default cursor to select salary details on an individual employee. This permitted the end user to switch between SELECT statements. If the user pressed the PF1 key without specifying a last name, the application selected all the rows. While scrolling through the rows (pressing the PF4 key), the user was also permitted to view each em-

ployee's salary history before viewing the next employee row. If the application did not use a named cursor to select employee rows, JAM/DB*i* would use the default cursor again, losing the user's place in first SELECT set when it issued the second SELECT statement.

Improving Efficiency

Before executing a SQL statement, the DBMS must prepare the statement. Preparation may include parsing the statement and declaring an engine-cursor. If a statement will be executed many times, declaring a cursor may improve the application's efficiency because the preparation is done only once, rather than each time the statement is executed. An application may declare some cursors upon start up or upon screen entry, and it may use function keys to call procedures which execute the named cursors.

4.3.3.

Error Processing

JAM/DB*i* provides two ways of managing errors in an application. The default method writes error messages to the status line, just as for JAM errors, and aborts the JPL procedure it was executing. The other method is for the developer to write and install an error handler which JAM/DB*i* will execute whenever a JAM/DB*i* error occurs.

An error handler written in JPL is installed with the statement

```
dbms ONERROR JPL procedure_name
```

An error handler written in C must be a prototyped function (i.e. installed in `pfuncs` in `funclist.c`) and is installed with the statement

```
dbms ONERROR C function
```

When a JAM/DB*i* error occurs, JAM/DB*i* will execute the installed error handler. JAM/DB*i* automatically passes arguments to the error handler—the text of the statement that failed, the engine name, and an error flag. The engine name is the name that was used to initialize the engine in `jmain.c`. The error flag equals 2.

The error handler is responsible for displaying any error messages. It may use `@dmretmsg` to display a JAM/DB*i* message, `@dmengerrmsg` to display an engine-specific error message, or it may examine the error codes `@dmretcode` and `@dmengerrcode` and display its own error messages.

The procedure's return code determines whether or not JPL continues or aborts the procedure it was executing.

Error handling is summarized in the figure below.

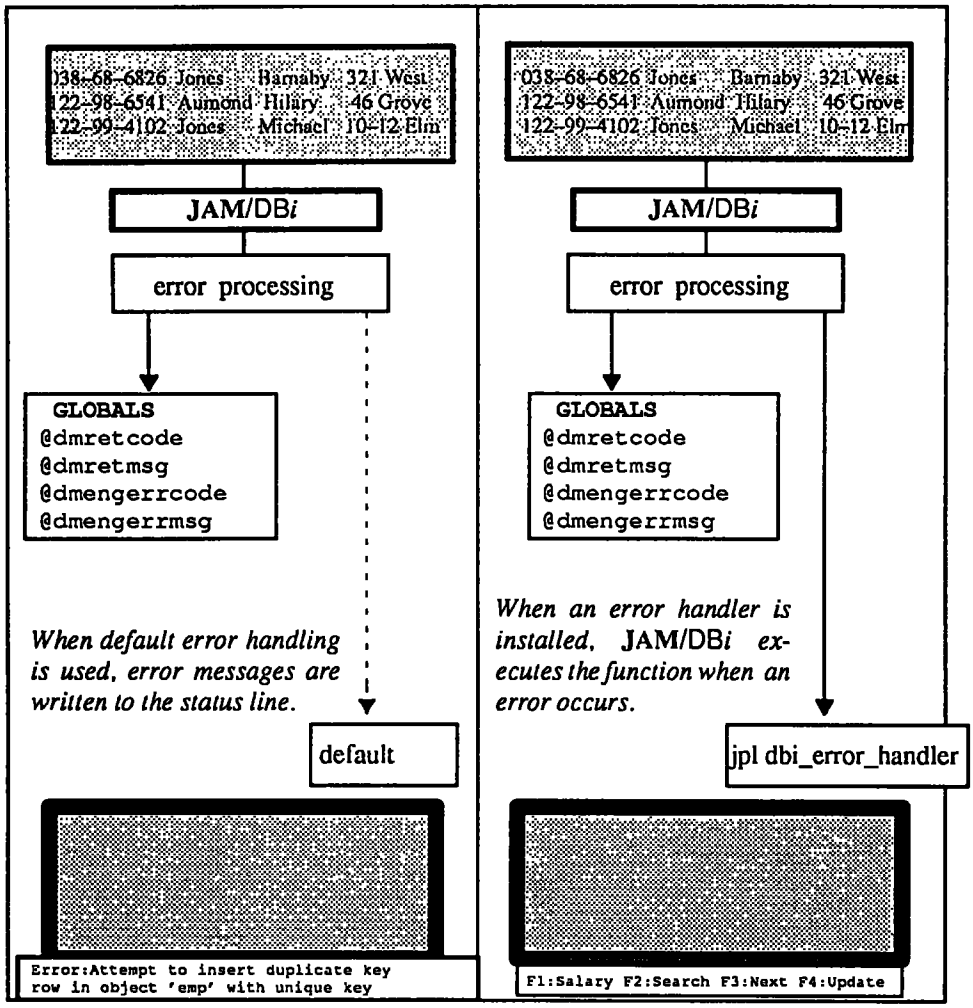
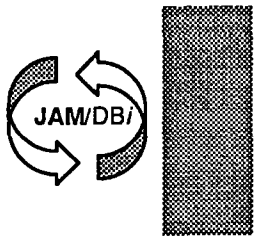


Figure 18: JAM/DB*i* Error Flow from the Database to JAM. The solid line shows the path used by the example.



Chapter 5.
JAM/DBi *Philosophy*

In this chapter, we address several features of JAM/DBi and suggest some development strategies.

5.1.
JAM/DBi FEATURES

JAM/DBi is a powerful tool for developing frontend applications and interfaces. The sections below discuss its prominent features.

5.1.1.
SQL–Based

SQL (Structured Query Language) is the standard for relational database languages. It is a tool which provides interactive users with a non-procedural, easy-to-use means of accessing databases and it assumes little or no programming skills. A key feature of JAM/DBi is that it uses the SQL syntax of the database you are using. You have complete access to all the features supplied by your DBMS. You do not need to learn a new syntax to use JAM/DBi because any SQL statement may be embedded in JPL and C hook functions. In JPL, a SQL statement is prefixed with the verb `sql` or associated with a declared cursor. In C, a SQL statement is passed as an argument to the JAM/DBi library function `dm_sql`.

As a result, JAM/DBi developers may create an entire frontend application simply using SQL and the JAM authoring tools.

5.1.2.
OS Portability

JAM/DB*i* is available on most operating system platforms. The JAM terminal and keyboard translation files provide all the hardware configuration needed by JAM/DB*i*. Developers customize the makefile distributed with JAM/DB*i* for software and operating system specifics.

5.1.3.
Vendor Independence

Vendor independence is an important feature of JAM/DB*i*. Since JAM/DB*i* is available for many popular relational databases, developers may choose a database for its data management capabilities while using JAM's powerful tools to create the frontend applications. In this way, developers are not limited by the vendor's frontend development tools.

In addition, JAM/DB*i* provides a standard means of moving applications from one database to another, with no changes to screens. If the two databases use different SQL syntax, however, developers may need to make some changes to SQL statements. Additional changes may be needed for differences in locking and transaction management on the two databases.

5.1.4.
Multi-engine Support

Some installations may maintain several databases, each with a DBMS supplied by a different vendor. JAM/DB*i* permits developers to access different engines in the same application. The user must have a JAM/DB*i* support routine for each DBMS product that the application will use.

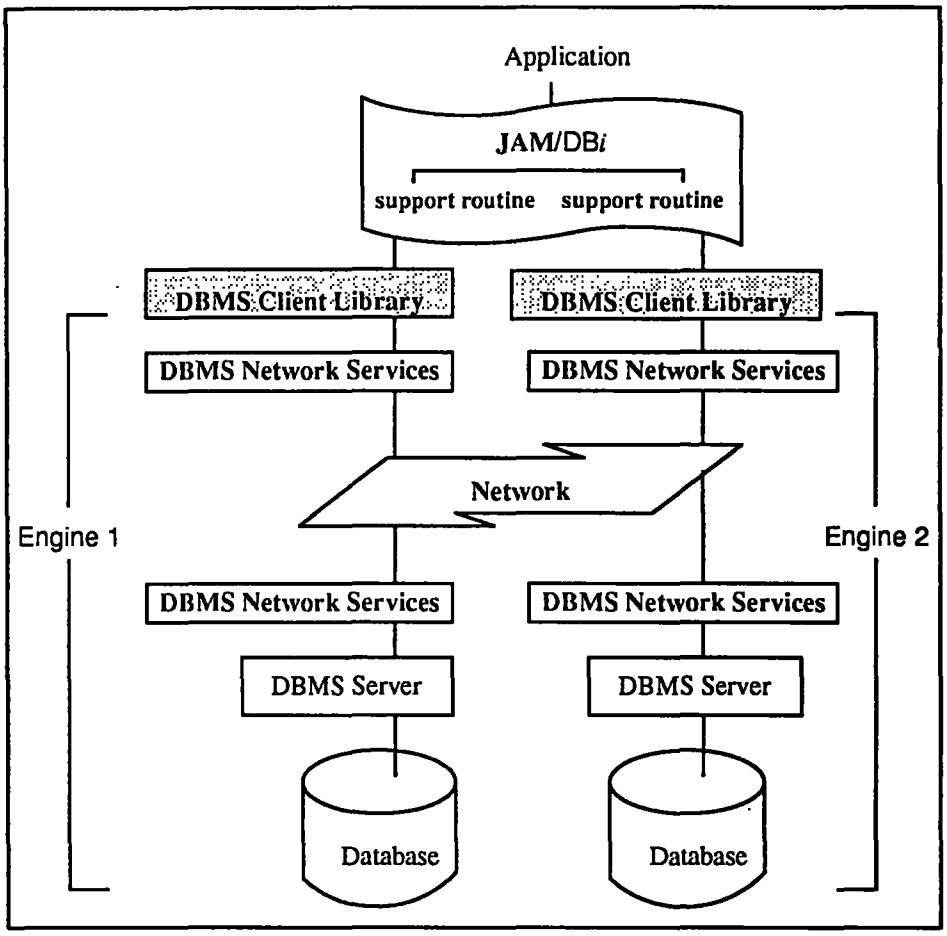


Figure 19: Components of JAM/DBi Architecture when using multiple engines.

5.1.5.

Multi–connection Support

Some engines permit multiple connections. This allows an application to have connections to multiple servers and databases of the engine. Connections are named, permitting the application to set a default connection and to switch between connections as it executes database operations.

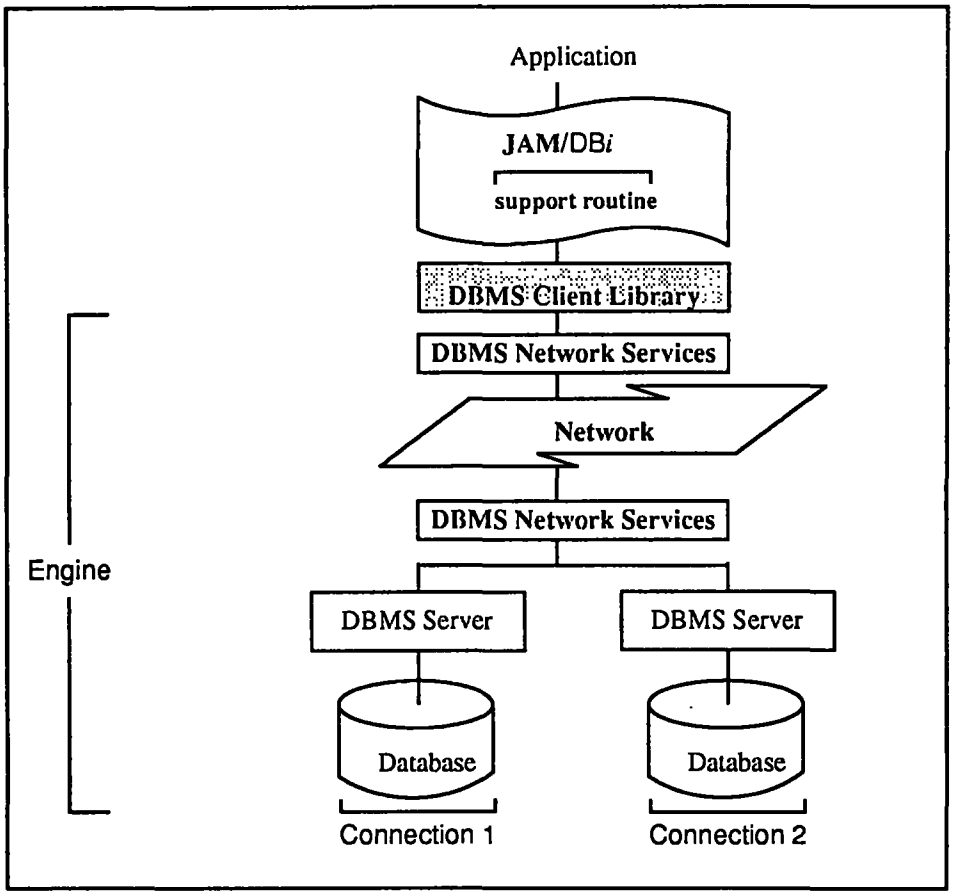


Figure 20: Components of JAM/DB*i* Architecture when using multiple connections.

5.1.6.

Prototyping

Developers using JAM/DB*i* may prototype an application with real links to a database without writing any third-generation programming code. Database functions may be simu-

lated by placing sample data on screens with JPL. Later, the the simulation code can be replaced with `sql` and dbms statements.

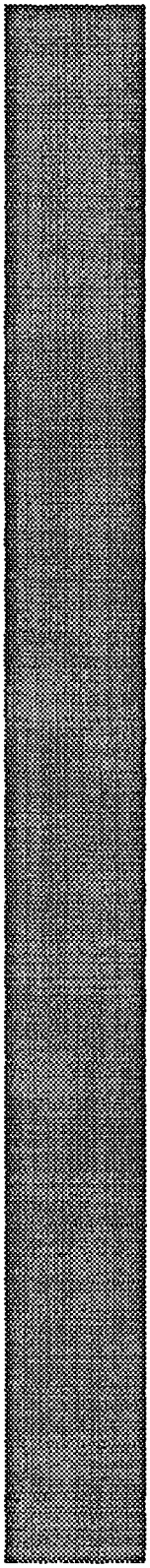
5.2. JAM/DBi DEVELOPMENT HINTS

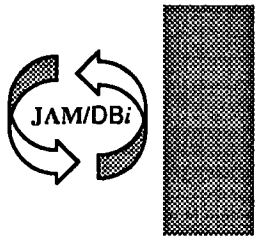
There are a few suggestions which developers should consider before developing an application.

- Execute `SELECT` statements when the target variables are on the active screen. Use the LDB just to pass a particular column value to another screen when necessary. In the sample application, two screens needed the values of the employee's social security number, first name, and last name. Rather than putting the target variable `ssn` in the data dictionary, the application defined `ssn` on the screen `empscrn` and defined `current_ssn` in the data dictionary. Therefore, `current_ssn` contains a value only when the application explicitly writes to the variable. By keeping only necessary column variables in the LDB, the developer reduces the amount of memory needed by the LDB, reduces the chances that the LDB will pass data to an unexpected target, and reduces the amount of application maintenance.
- Use an error and/or exit handler to process error and status information. Not only does this reduce the amount of code in the application, it also ensures consistent error handling throughout the application.

Appendix C. covers these topics in more detail.

Developer's Guide





Chapter 6.
Introduction to Development

This document is intended for JAM/DB*i* developers. We discuss the development and creation of executable JAM/DB*i* programs using developer-written hook functions to access and manipulate a database.

We assume that the reader is familiar with JAM. JAM/DB*i* developers should see the JAM *Author's Guide* for information on using the Screen Editor, Keyset Editor, and the Data Dictionary Editor. They should see the *JPL Guide* for information on writing and storing JPL procedures. They should see the *Programmer's Guide* for information on installing C hook functions in the application function list and for customizing the source modules, `jmain.c` or `jxmain.c`.

In addition, developers should review the JAM *Development Overview* and the JAM/DB*i* *Development Overview* before proceeding. These sections discuss the architectural components and the control flow of JAM and JAM/DB*i*.

6.1.
SQL VARIANTS

SQL is an evolving standard in the database industry and there are numerous SQL-based products on the market today. At this writing JAM/DB*i* supports more than ten vendors' SQL-based products. Each of these vendors implements aspects of SQL differently. For example, some engines permit the use of only single quotes around literals in query statements. Other engines permit the use of either single or double quotes. Engines often have different rules for the use of case and special characters in variable names. JAM/DB*i* provides features to assist developers with these differences. Developers may use the colon-plus preprocessor to format values for a particular DBMS engine before inserting them in database columns. They may control case handling by setting the engine's case flag at initialization.

The obvious advantage is ease of use. JAM/DB*i* provides access to almost all functions supported by the vendor, without changes in command syntax. Developers concerned with DBMS portability, however, must use a compatible SQL syntax. For example, the SQL syntax of most vendors includes a subset of ANSI-compliant SQL commands. The syntax of these commands is usually portable.

The *Developer's Guide* discusses concepts common to all supported engines. For this reason, we do not emphasize any particular implementation of SQL. Any SELECT, INSERT, UPDATE, or DELETE statement in the examples is used only to clarify concepts. When using the concept in an actual application, use the SQL syntax of the DBMS.

6.2. JAM/DB*i* COMMANDS

Developers may execute JAM/DB*i* functions from JPL statements and C language function calls. JAM/DB*i* distinguishes between two types of database commands. In JPL, database commands are executed with either the command `sql` or the command `dbms`. Similarly in C, database commands are executed with the functions `dm_sql` or `dm_dbms`.

The `sql` variants execute statements that may be given in the interactive query language of the database. They include CREATE, DROP, SELECT, INSERT, UPDATE and DELETE.

The `dbms` variants execute the following types of functions:

- Statements not needed or not supported in the database's interactive query language. (i.e., LOGON, DECLARE CURSOR, CONTINUE)
- Statements to customize the JAM/DB*i* environment. These include error trapping and directing output to a file or an array occurrence.
- Vendors' "extended" SQL functions. These functions are non-standard enhancements to SQL (e.g., browse, control execution of a stored procedure, etc.).
- SQL statements to be executed under the control of explicitly declared cursors.

Actually, any SQL statement may be executed with a `dbms` command. This is done in two steps: a cursor is declared and associated with the SQL statement, and then the cursor is executed. Developers may use the "short-cut" command `sql` to execute simple queries in a single step. For example,

```
dbms DECLARE item_cursor CURSOR FOR \
   SELECT description, price FROM products \
   WHERE code = :+code
dbms WITH CURSOR item_cursor EXECUTE
```

fetches the same rows as

```
sql SELECT description, price FROM products \
   WHERE code = :+code
```

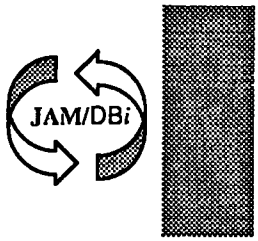
6.2.1.

JPL versus C

The colon preprocessor has always been a powerful incentive to use JPL rather than C for JAM/DB*i* functions. Release 5 makes two improvements to the colon preprocessor: it provides a special form for formatting database values, and it performs colon preprocessing on the arguments of `dm_dbms` and `dm_sql`, the library functions for executing database commands.

The decision to use JPL or C is left to the developers' discretion. Developers should know that they may execute any SQL statement from either language, and they may use either or both languages in an application. JPL procedures may be executed without compilation.

Most of the examples in this guide use JPL.



Chapter 7.
Access and Execution

In this chapter we discuss how an application accesses and queries a database. We discuss the following topics:

- Initializing one or more engines – the application tells JAM/DB*i* which engines (i.e., vendor products) it will use. (Section 7.1.)
- Connecting to a server and database – the application connects to a server where an initialized engine is running. (Section 7.2.)
- Using cursors – the application uses a default or named cursor to execute an operation on a connection. (Section 7.3.)

7.1. INITIALIZING ONE OR MORE ENGINES

An *engine* is a DBMS product. It is identified by a specific vendor and version. For example, SYBASE 4.0, ORACLE 6.0, and ORACLE 5.1 are three distinct engines. JAM/DB*i* is distributed with an object file containing a support routine for a particular engine. The support contains all the vendor-specific code necessary for executing database operations with JAM/DB*i*.

JAM/DB*i* permits an application to access one or more engines. The application must have a support routine for each engine, and it must initialize an engine before opening a connection or a executing a query on the engine.

7.1.1. Initializing an Engine in `dbiinit.c`

A call to initialize one or more engines may be put in the JAM/DB*i* source module `dbiinit.c`. A sample `dbiinit.c` is distributed with JAM/DB*i*. The file,

1. makes a function declaration for one or more support routines
2. describes the engine initialization in the structure `vendor_list`

`vendor_list` appears like the following,

```
static vendor_t vendor_list[] =
{
    {"engine", support_routine, case_flag | error_flag, (char *) 0},
    {(char *)0, (int (*)()) 0, (int) 0, (char *) 0}
};
```

The name for ***engine*** is chosen by the developer. If an application uses two or more engines, the application will use the mnemonic ***engine*** to tell JAM which DBMS to use. Most of the examples in the guide use a vendor name as the mnemonic, for example `sybase` or `oracle`, but any character string that is not a keyword is valid. Keywords are listed in Appendix A.

The name of ***support_routine*** is documented in the distributed `dbiinit.c`. The name is usually in the form `dm_`***vendor***`sup` where ***vendor*** is an abbreviated vendor name. Some examples are

- `dm_intsup`
- `dm_orasup`

- dm_sybsup

case_flag sets the case-handling feature of JAM/DB*i*. It determines how JAM/DB*i* uses case to map column names to JAM variables when executing a SELECT. The options are

- DM_PRESERVE_CASE — Use case exactly as returned by the engine.
- DM_FORCE_TO_UPPER_CASE — Force all column names returned by an engine to upper case. The developer should use upper case when naming JAM variables.
- DM_FORCE_TO_LOWER_CASE — Force all column names returned by an engine to lower case. The developer should use lower case when naming JAM variables.
- DM_DEFAULT_CASE — Usually defaults to DM_PRESERVE_CASE. Another default value may be set by JYACC in the support routine.

For example, ORACLE returns all column names in upper case. If DM_PRESERVE_CASE is set, JAM/DB*i* will look for 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 it may return column names in upper, lower, or mixed cases. To map SYBASE columns to single case JAM variables, set the case flag to DM_FORCE_TO_UPPER_CASE or DM_FORCE_TO_LOWER_CASE.

error_flag determines which error messages are displayed by the default error handler. This flag is "or–ed" with the case flag. The options are

- DM_DEFAULT_DBI_MSG — The default error handler displays engine-independent error messages when an error occurs. These messages are defined in the JAM message file.
- DM_DEFAULT_ENG_MSG — The default error handler displays engine-dependent error messages when an error occurs. These messages are supplied by the engine.

If neither flag is used, the default is DM_DEFAULT_DBI_MSG.

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

If the DBI subsystem is installed (i.e., its macro is set to 1 in `jmain.c` or by a compiler directive), `jmain` (or `jxmain`) will call the JAM/DB*i* library function `dm_init` for each support routine in the list.

If the initialization is successful, ***support_routine*** returns zero. In some cases ***support_routine*** may reject the initialization and return an error code. In these cases, there may be insufficient memory, the engine may not be installed, or the application may have initialized the same support routine more than once. If such an error occurs when executing `jmain`, JAM will display an error message and terminate.

7.1.2.
Initialization Procedure

As a part of initialization, JAM/DB*i* calls the support routine for information on the particular DBMS. For each ***engine***, JAM/DB*i* has information on the following

- the engine's capabilities (e.g., whether the engine can execute stored procedures or support multiple connections)
- the required formatting for character and null strings being inserted into a table
- the default for case handling

In addition, JAM/DB*i* sets up some structures at initialization, including structures for tracking the number and names of all connections on an engine.

7.1.3.
Setting the Default Engine

The application may connect to any initialized engine.

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

```
DBMS ENGINE engine
```

or sets a *current engine* for a statement with the clause `WITH ENGINE`. An application accessing multiple engines must reset the default or current engine when declaring connections to the different engines. Once a connection is declared, the default connection determines the default engine.

7.2.
CONNECTING TO A DATABASE SERVER

Before performing operations on database tables, JAM/DB*i* must connect to a DBMS server with the statement

```
dbms [WITH ENGINE engine] DECLARE connection CONNECTION \
   FOR OPTION argument [OPTION argument]
```

Different engines support different options. Please see the DBMS-specific *Notes* in this document for a list of the valid options.

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

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

- a connection name
- a pointer to engine information
- 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/DB*i* is able to close all open cursors on the connection)

If no engine is named, the connection is declared for the default engine.

The statement

```
dbms CLOSE CONNECTION connection
```

logs off and closes the connection.

7.2.1.
Connections to Multiple Engines

If an application is using two or more engines, a connection may be declared for each engine. A default connection may be set with the command

```
dbms CONNECTION 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
sql SELECT * FROM emp WHERE last = :+last
```

In the example, connections are declared on the engine `sybase` and the engine `oracle`. The connection `sybcon` is chosen as the default. Therefore, JAM/DB*i* performs the `SELECT` on the connection `sybcon` and uses the support routine of `sybcon`'s engine to execute the query.

The `WITH CONNECTION` clause specifies a connection to be used for a single statement, overriding the default connection. For example,

```
sql WITH CONNECTION oracon SELECT * FROM sales
```

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

7.2.2.

Multiple Connections to a Single Engine

Some engines permit two or more simultaneous connections. See the DBMS-specific *Notes* in this document for information on your engine. Developers who wish to take advantage of this feature on a valid engines should declare a named connection for each session on the engine.

```
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
```

If this is the second or later connection on the engine, and the engine supports multiple connections, the support routine opens the additional connection and JAM/DB*i* keeps a count of the number of active connections for the engine. If the engine does not support multiple connections or the connection name is not unique, JAM/DB*i* returns the error `DM_ALREADY_ON`.

The application may close all connections by executing `DBMS CLOSE CONNECTION` for each declared connection or it may close all connections on an engine or all engines by executing

```
dbms [WITH ENGINE engine] CLOSE_ALL_CONNECTIONS
```

7.3. USING CURSORS

A *cursor* is a SQL object associated with a specific query or operation. JAM/DB*i* stores information on each cursor. This includes,

- the cursor's name
- the cursor's connection
- any cursor attributes assigned with the commands `DBMS ALIAS`, `DBMS CATQUERY`, `DBMS FORMAT`, `DBMS OCCUR`, `DBMS START`, `DBMS STORE`, and `DBMS UNIQUE`
- other operation–specific information (e.g., the number of rows to fetch, information on target variables or binding parameters, etc.)

Cursors are not JAM variables, and they do not follow the scoping rules of JAM variables. When a cursor is declared, JAM/DB*i* 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/DB*i* frees the structure when the cursor is closed.

Every connection has one or two default cursors which JAM/DB*i* automatically creates. An application may also declare named cursors on a connection. A JAM/DB*i* 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 `sql` or the library function `dm_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 section 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 not covered here but in Chapters 8. and 9.

7.3.1. Using the Default Cursor

For most engines, JAM/DB*i* automatically declares two default cursors—one for `SELECT` statements and one for non-`SELECT` statements such as `UPDATE`. In a few cases, the engine's

standard is a single default cursor and JAM/DB*i* will declare one default cursor. On such engines, an additional option, CURSORS, is supported in the engine's DECLARE connection statement. It permits the developer to choose between one or two default cursors for the connection. See the DBMS-specific *Notes* in this document for more information.

A default SELECT cursor is associated with a particular connection, namely the connection in effect when a SELECT statement is executed. For example,

```
dbms CONNECTION c2
dbms WITH CONNECTION c1 \
   SELECT code, region FROM sales WHERE sales > 999.99
sql UPDATE sales SET code = :+code WHERE region = :+new
```

The first statement sets the default connection. The second statement uses WITH CONNECTION to set c1 as the current connection for the SELECT statement. In the last statement, no connection is specified for the UPDATE statement. Therefore, JAM/DB*i* uses the default connection c2.

7.3.2.

Using a Named Cursor

A developer may create one or more named cursors to access and manipulate data. The sequence is the following:

- 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/DB*i* uses the default connection.

```
dbms [WITH CONNECTION connection] DECLARE cursor CURSOR \
   FOR SQLstmt
```

For example,

```
dbms DECLARE customer_cur CURSOR FOR \
   SELECT * FROM directory WHERE lname = :+lname
```

This statement is a declaration statement. JAM/DB*i* does not pass the query to the DBMS. Instead it parses the query, performing any specified colon expansion. Colon expansion is not repeated when the cursor is executed.

Executing a Cursor

Once a cursor has been created, the statement

```
dbms WITH CURSOR cursor_name EXECUTE
```

executes the SQL statement associated with ***cursor_name***. For the examples used above, the statement

```
dbms WITH CURSOR customer_cur EXECUTE
```

executes the SQL statement `SELECT * FROM directory WHERE lname =` ***value of lname when cursor was declared.*** If qualifying rows are found, the database will return them now to JAM/DB*i*.

If the SQL statement is a `SELECT` statement that retrieves more rows than will fit on the screen, the statement

```
dbms WITH CURSOR cursor_name CONTINUE
```

continues the previous `EXECUTE` for ***cursor_name*** by fetching the next screenful of records from the `SELECT` set.

Executing a Cursor with Parameters

Parameters may be passed with the statement `DBMS EXECUTE`. The syntax is the following:

```
dbms [WITH CONNECTION connection] DECLARE cursor CURSOR \
   FOR SQL statement
dbms [WITH] CURSOR cursor EXECUTE USING var1 [, var2...]
```

There is a one-to-one mapping between parameters in ***SQL statement*** and the ***var*** values in the `USING` statement. In a `DECLARE CURSOR` statement for any engine, JAM/DB*i* interprets `::`***parameter*** as a binding parameter. For example,

```
dbms WITH CONNECTION c1 DECLARE x_cursor CURSOR \
   FOR SELECT * FROM sales WHERE cost = ::parm

dbms WITH CURSOR x_cursor EXECUTE USING newcost
```

Note that the use of parameters is different than the use of colon preprocessing when declaring a cursor. When the colon preprocessor is used, column values are supplied when the cursor is declared. To use different values, the cursor must be redeclared before it is executed. When binding is used, the application supplies column values each time it executes the cursor.

If an engine uses another syntax for binding parameters, JAM/DB*i* will also support it.

This topic is covered in detail in Section 8.2.

Note to Developers Using 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 curl CURSOR FOR SELECT * FROM books
dbms CONNECTION oracon
dbms WITH CURSOR curl EXECUTE
sql UPDATE ....
```

When cursor curl is declared JAM/DB*i* associates it with the default connection sybcon1. Although the default connection is changed to oracon before the cursor is executed, the connection associated with curl does not change. When the cursor is executed, the JAM/DB*i* performs the SELECT on connection sybcon. The default connection oracon performs the subsequent UPDATE.

Modifying or Closing a Cursor

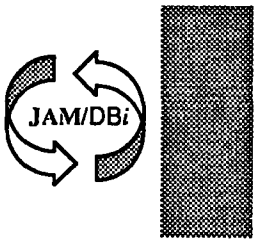
A cursor may be redeclared for another SQL statement. For example,

```
DBMS DECLARE abc CURSOR FOR \
   SELECT order_id, total FROM newsales \
   WHERE total > :+cost
DBMS WITH CURSOR abc EXECUTE

DBMS DECLARE abc CURSOR FOR \
   SELECT * FROM directory WHERE dept = 'Sales'
DBMS WITH CURSOR abc EXECUTE
```

JAM/DB*i* provides several commands for changing the default behavior for a cursor associated with a SELECT statement. The commands are DBMS ALIAS, DBMS CATQUERY with DBMS FORMAT, DBMS OCCUR, and DBMS START. They are discussed in Chapter 9. Here we note that these settings are not lost when a cursor is redeclared, but only when the cursor is closed.

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

```
dbms CLOSE CURSOR cursor_name
```



Chapter 8.
Data Flow from JAM

This chapter discusses how JAM/DB*i* 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*** and :+***variable***.
- Parameters: binding values to SQL parameters when executing a named cursor. Their form is ::***variable***.

8.1. COLON PREPROCESSING

JAM supports two types of colon preprocessing,

- :var Standard colon preprocessing, and
- :*var Re-expanded colon preprocessing.

Both methods are described in the *JPL Guide* in Volume II of JAM. One or more colon variables may appear almost anywhere in a `sql` or `dbms` statement. There are two exceptions.

The first word in the statement may not be colon-expanded. Therefore, the statements

```
:verb SELECT * FROM students
:command EXECUTE cursor1
```

are both illegal. JPL must know the command word to perform syntax checking and compilation before executing a JPL statement.

Colon expansion is not permitted in the `WITH ENGINE` or the `WITH CONNECTION` clause. Therefore,

```
dbms :eng_str DECLARE c1 CONNECTION FOR USER :uname
sql WITH CONNECTION :cname SELECT * FROM students
```

are also both illegal. JPL must know which engine or connection is in use before performing any colon processing.

In addition to the standard forms, JAM/DB*i* supports special forms of colon pre-processing for values sent to a database. The forms are

- :+var Database colon preprocessing for column values (colon-plus)
- :=var Database colon preprocessing for operator and column values (colon-equal)

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.

8.1.1. Colon-plus Processing

Before colon preprocessing a statement, JPL determines which engine to use. If executing a `sql` or `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. In other JPL statements, such as `cat`, JPL always 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 source. If ***variable*** has a null edit and its value is the null edit's string, the colon-plus processor replaces the value with the engine's null value. If it does not have a null edit, or does not contain the null edit string, the processor determines the variable's JAM *type*. The term JAM *type* refers to a classification of JAM field characteristics used by the library function `sm_ftype`, the colon-plus processor, and JAM/DB*i* routines for binding. The JAM types are
   - `DT_CURRENCY`
   - `DT_DATETIME`
   - `DT_YESNO`
   - `FT_CHAR`
   - `FT_DOUBLE`
   - `FT_FLOAT`
   - `FT_INT`
   - `FT_LONG`
   - `FT_PACKED`
   - `FT_SHORT`
   - `FT_UNSIGNED`
   - `FT_VARCHAR`
   - `FT_ZONED`
3. If the JAM type is `DT_DATETIME`, `FT_CHAR`, or `FT_VARCHAR` the processor formats the value according to engine-specific rules, usually enclosing the string in quote characters. For the 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 steps are described in full below.

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
- fields on the current screen
- LDB variables[3]

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.

For more information on variables and scope, see the *JPL Guide*.

Step 2. Determine the Variable's JAM Type

If the variable is a field or LDB entry that has a null edit, and 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, if field named `address` had a null field edit, the Screen Editor window could appear as the following:

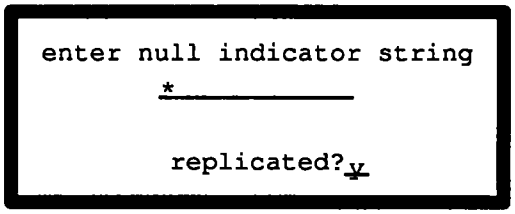


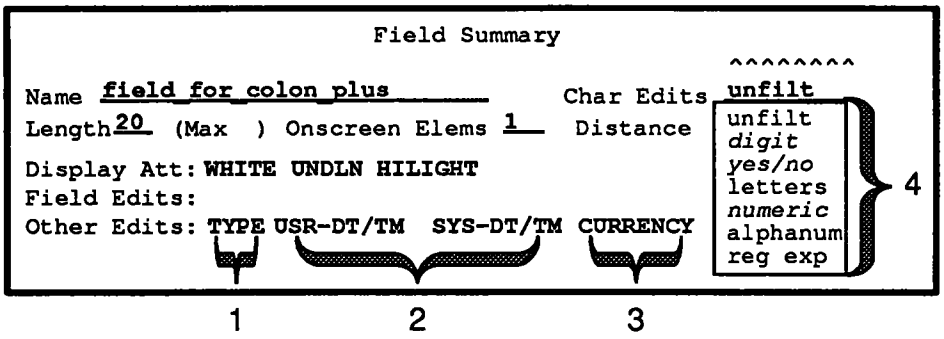
Figure 21: Null field edit window in JAM Screen Editor.

If the user or program does not enter text in the field named `address`, the field is null and JAM will display the string, ******* as the field contents. JAM/DB*i* would convert the string ******* to NULL (i.e., the value of the engine's null string) before passing it to a DBMS.

If the variable does not have a null edit, or its value does not equal its null edit string, the processor calls a routine to examine field characteristics and determine the variable's JAM *type*.

3. 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.

A field or LDB variable has exactly one JAM type. Since a variable may have more than one of the qualifying PF4 characteristics, JAM uses some precedence rules when assigning the JAM type.



Summary Keyword	Setting of Field Characteristic (PF4 menu in draw mode)	Submenu Option	JAM Type
TYPE	type (C types for structures)	char string int unsigned int short int long int float double zoned dec. packed dec.	FT_CHAR FT_INT FT_UNSIGNED FT_SHORT FT_LONG FT_FLOAT FT_DOUBLE FT_ZONED FT_PACKED
USR-DT/TM SYS-DT/TM	misc. edits	date or time	DT_DATETIME
CURRENCY	misc. edits	currency	DT_CURRENCY
Char Edits	char edits	digits only yes/no field numeric	FT_UNSIGNED DT_YESNO FT_DOUBLE

Figure 22: Field Summary Screen (PF5 in draw mode). Use the summary screen to determine a field's JAM type. A TYPE edit has the highest priority, then a date time edit, then a currency edit, and finally a character edit. A variable with any other edits has the JAM type FT_CHAR.

C record types are assigned with the `type` option on the PF4 key menu. For clarity, we call these types *C types*. To assist developers using utilities such as `f2struct`, JAM automatically assigns a default C type to each field. Developers may also explicitly set a C type. JAM/DB*i* ignores C types assigned by default; it only uses those assigned explicitly by a

developer. The field summary screen is an easy way of checking whether or not JAM/DB*i* will use the variable's C type. If the word `TYPE` is shown on the `Other Edits` line of the field summary window, and the type is not `omit`, JAM/DB*i* will use it to assign a JAM type.

Otherwise, JAM examines the miscellaneous edits; a date-time or currency edit will provide a JAM type. If the variable does not have a date-time or currency edit, JAM examines the variable's PF4 `char edits`. An edit of `digits only`, `yes/no field`, or `numeric` will provide a JAM type. For all other field and LDB variables, and for all JPL variables, JAM assigns `FT_CHAR` as the JAM type.

Beware of C type edits that may conflict with other edits. For example, if a field had a type edit `int` and a `date-time edit`, its JAM type would be `FT_INT`. The Screen Manager would enforce the date-time format for user entry but JAM/DB*i* would not convert the date-time string into a format the engine would recognize.

Note: developers may also use `sm_ftype` 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 JAM/DB*i* determines a variable's JAM type, it uses the classification to perform any necessary formatting and returns the formatted text to JPL.

`DT_DATETIME` Variable

If 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/DB*i* formats the date as

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

If the engine is using the SYBASE support routine, however, JAM/DB*i* formats the date as

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

Some engines support more than one datatype for date-time columns. Please see the engine-specific *Notes*.

FT_CHAR Variables

If 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 2 (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/DB*i* would format the field value

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

to

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

JAM/DB*i* 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, please see the *Notes* section in this document.

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. See the JAM *Programmer's Guide* for more information on sm_strip_amt_ptr. The colon preprocessor does not use precision edits when formatting numeric values.

For example, JAM/DB*i* would format

```
$500,000.00
```

as

```
500000.00
```

JAM/DB*i* 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 field's C type to `char`.

See the engine-specific *Notes* for additional information.

8.1.2.

Colon-equal Processing

To specify a NULL value in a search criteria, most engines require the syntax

```
SELECT column_list FROM table WHERE column IS NULL
```

To permit endusers to select rows where a column value is either known or unknown (i.e., NULL), use the colon-equal processor. For example,

```
sql SELECT * FROM emp WHERE zip :=zip
```

If `zip` is a character field with the null edit

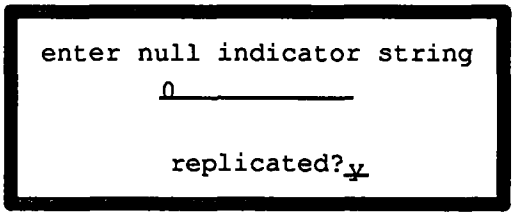


Figure 23: Null field edit window in JAM Screen Editor.

JAM/DB*i* would format the value

```
10038
```

as

```
= '10038'
```

thus executing

```
SELECT * FROM emp WHERE zip = '10038'
```

It would format the field's "null" value

```
00000
```

as

```
IS NULL
```

thus executing

```
SELECT * FROM emp WHERE zip IS NULL
```

8.1.3.

Examples

A Field with Default Characteristics

If the current screen has a field named `last` with no field, miscellaneous or type edits, and a character edit `unfilt`, its field summary screen would appear as

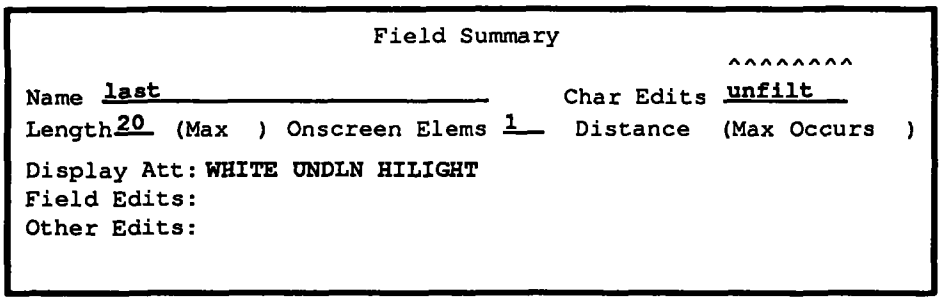


Figure 24: Field Summary Screen. With these edits, JAM type = `FT_CHAR`.

Since the field does not have any of the field characteristics listed in Figure 22 on page 65, JAM type = `FT_CHAR`. If the field `last` contained the text `D'Angelo` when the following were executed,

```
sql SELECT * FROM employee WHERE last = :+last
```

JAM/DB*i* would pass the query

```
SELECT * FROM employee WHERE last = 'D''Angelo'
```

If the field `last` were empty, JAM/DB*i* would pass the empty string, not the null string,

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

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

A Variable with a Date-time Edit and a Null Edit

If the current screen contains a field `hiredate` with a null field edit string `00/00/00`, a date-time edit `MON2/DATE2/YR2` for a user-specified date, and character edit of `digits only`, its summary screen would appear as

```
                    Field Summary
                                                ^^^^^^^^
Name hiredate                        Char Edits digit
Length8  (Max  ) Onscreen Elems 1    Distance  (Max Occurs  )
Display Att: WHITE UNDLN HILIGHT
Field Edits:
Other Edits: USR-DT/TM NULL
```

Figure 25: Field Summary Screen. For this field, JAM type = `DT_DATETIME`.

Assume that back slash characters are saved with the field as embedded punctuation. Since a date-time edit has a higher precedence than a character edit, the JAM type for this field is `DT_DATETIME`. If the user entered the date `12/31/91` and executed the following function,

```
sql WITH CONNECTION oracle_conn \
   INSERT INTO employee (last, hiredate) \
      VALUES (:+last, :+hiredate)
```

and the engine, for example, were ORACLE, JAM/DB*i* would pass the statement

```
INSERT INTO employee (last, hiredate) VALUES \
      ('D''Angelo', \
       TO_DATE('31121991 000000', 'ddmmyyyy hh24miss'))
```

to the engine.

If the user did not change the text in the field `hiredate`, so that its contents were `00/00/00`, JAM/DB*i* would pass the statement

```
INSERT INTO employee (last, hiredate) \
   VALUES ('D''Angelo', NULL)
```

to the engine.

A Variable with a Digits Only Character Edit and a C-Type char string Edit

Very often it is useful to use the `digits only` character edit on fields that accept values such as a social security number, zip code, or telephone number. If this is the only edit on the field, the colon-plus processor will format the field's value as an unsigned integer, removing embedded punctuation and leading zeros. However, if the developer resets the C-type edit to `char string`, the colon-plus processor will format the field's contents as a character string, preserving embedded punctuation and leading zeros.

If the current screen contains a field `zip_code` with a character edit of `digits only` and a C type of `char string`, its summary screen would appear as

```
                    Field Summary
                                                ^^^^^^^^
Name zip_code                        Char Edits digit
Length5 (Max  ) Onscreen Elems 1     Distance  (Max Occurs  )
Display Att: WHITE UNDLN HILIGHT
Field Edits:
Other Edits: TYPE
```

Figure 26: Field Summary Screen. For this field, JAM type is set according to the value of TYPE. If TYPE is "char string" JAM type = `FT_CHAR`.

For example, if a user entered `00912` in the field `zip_code` and executed the following function,

```
sql SELECT * FROM marketing WHERE zip = :+zip_code
```

JAM/DB*i* would pass the query

```
SELECT * FROM marketing WHERE zip = '00912'
```

to the DBMS.

Note that if the developer assigned `digit only` to the field, but did not reset the C type, JAM/DB*i* would pass the query

```
SELECT * FROM marketing WHERE zip = 912
```

8.2.
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/DB*i* internally supports cursors with parameters.

When JAM/DB*i* executes a `DECLARE CURSOR` statement, it scans the statement for parameters. For all engines, JAM/DB*i* recognizes

`::`***parameter***

to be a parameter.[4] If JAM/DB*i* 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 emp WHERE last = ::xyz

dbms DECLARE b_cursor CURSOR FOR \
   INSERT INTO acc VALUES (::ss, ::sal, ::exmp)

dbms DECLARE c_cursor CURSOR FOR \
   UPDATE emp SET street=::street, city=::city, \
   st=::st, zip=::zip WHERE ss=::ss

dbms DECLARE d_cursor CURSOR FOR \
   DELETE newsales WHERE custid=::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.

4. 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/DB*i* also recognizes these engine-specific forms.

A value for a parameter is supplied in the USING clause of an EXECUTE statement,

```
dbms WITH CURSOR cursor EXECUTE USING arg [, arg...]
```

JAM/DB*i* 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/DB*i* 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 acc VALUES (::p1, ::p2, ::p3)
....
dbms WITH CURSOR b_cursor EXECUTE USING ss, sal, exmp
```

Values and parameters may also be bound explicitly by name,

```
dbms DECLARE b_cursor CURSOR FOR \
   INSERT INTO acc VALUES (::p1, ::p2, ::p3)
....
dbms WITH CURSOR b_cursor EXECUTE \
   USING p3=exmp, p1=ss, p2=sal
```

Note that `p3`, `p1`, and `p2` are not JAM variables but `exmp`, `ss`, and `sal` are. JAM/DB*i* uses the values of `exmp`, `ss`, and `sal` to execute the INSERT. To supply a literal value to the INSERT, put the value in quotes,

```
dbms WITH CURSOR b_cursor EXECUTE \
   USING p1=ss, p2=sal, p3="0"
```

JAM/DB*i* 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/DB*i* 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.

8.2.1.
Parameter Substitution and Formatting

An ***arg*** in a USING clause may be either

- a quoted string, or
- a JAM variable

Colon-plus processing is not necessary because JAM/DB*i* automatically formats the value of parameter variables. If the variable is an array name, an occurrence number may be given. If no occurrence is given, JAM/DB*i* concatenates all the non-empty occurrences in the array, separating the occurrences with a single space. Substrings are not permitted.

For each cursor, JAM/DB*i* maintains binding information. When a cursor's statement uses parameters, JAM/DB*i* stores the names of the parameters. When a cursor is executed, JAM/DB*i* 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/DB*i* uses a data structure to store the formatted text and JAM type of ***arg***. If ***arg*** is not quoted, JAM/DB*i* 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 has a null edit and contains the null edit string.

If `ftype=DT_DATETIME`, JAM/DB*i* calls the support routine to convert the value to a binary date-time value. See the discussion of DT_DATETIME on page 66 for more information.

No processing is done on the values of FT_CHAR variables or quoted strings.

For all other types, JAM/DB*i* 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 emp WHERE name=::p1 or ss=:p2

# newname and ss_number are LDB variables
dbms WITH CURSOR x EXECUTE \
   USING p1=newname, p2=ss_number

# code is a JPL variable containing the text "ss_number"
# and ss_number is a field on the current screen
dbms WITH CURSOR x EXECUTE USING p1='Jones', p2=:code

# name and ss_number are field arrays. i is a JPL variable
dbms WITH CURSOR x EXECUTE \
   USING p1=name[i], p2=ss_number[i]
```

Examples

If the current screen contained a field named `total` with a currency edit and character edit of `numeric` its summary screen would appear as

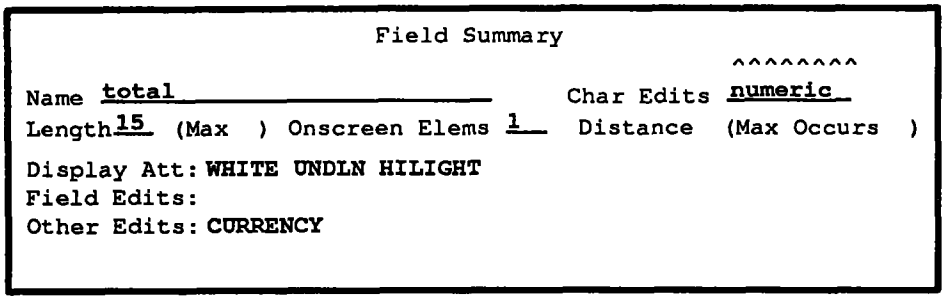


Figure 27: Field Summary Screen. For this field, `ftype = DT_CURRENCY`.

If the user entered the total `$9,499.99` and executed the following statements:

```
dbms DECLARE sales_cursor CURSOR FOR \
   SELECT * FROM orders WHERE total > ::x
...
dbms WITH CURSOR sales_cursor EXECUTE USING x=total
```

the DBMS would execute

```
SELECT * FROM orders WHERE total > 9499.99
```

If the current screen contained a field named `description` with a `null field` edit and a `word wrap` edit, its summary screen would appear as

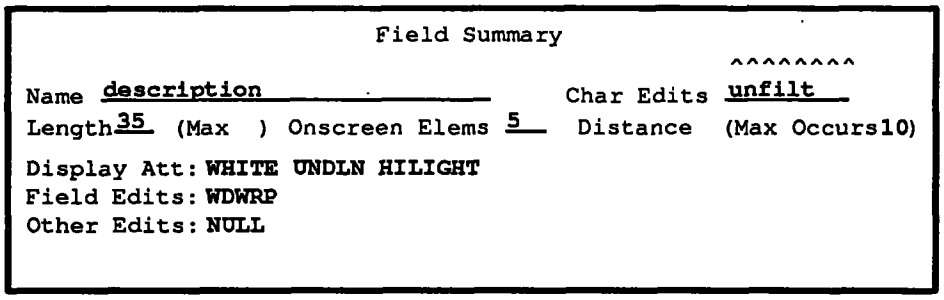


Figure 28: Field Summary Screen. With these edits, `ftype = FT_CHAR`.

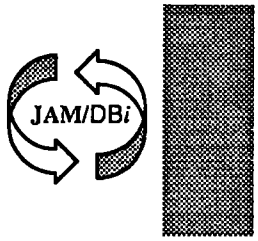
If the user executed the following statements:

```
dbms DECLARE ins_cursor CURSOR FOR \
   INSERT INTO products (description) VALUES (::p1)
...
dbms WITH CURSOR ins_cursor EXECUTE USING description
```

when the word wrapped array were empty, the DBMS would execute

```
INSERT INTO products (description) VALUES ('')
```

If, however, the array contained text, JAM/DB*i* would concatenate the non-empty occurrences into one long string which the DBMS would insert into the column `description`.



Chapter 9.
Data Flow from a Database

A JAM/DB*i* application receives two types of information from a database:

- data requested by a SELECT statement
- a count of the rows fetched for a SELECT statement
- error and status codes from an engine and from JAM/DB*i*

The rest of the chapter discusses how this information flows from one or more databases to variables in a JAM application. The first part discusses the destination and format of data returned by SELECT statements. The second part discusses the global JAM/DB*i* variables for status and error data.

In addition to the two types of information described above, an application may 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, the topic is covered in the *Notes* section of this document.

9.1.

DATA FETCHED BY SELECT

When a SELECT statement is passed to an engine, JAM/DB*i* performs several steps before transferring data to JAM variables.

1. JAM/DB*i* counts the number of columns in the query and records information on each column's name, length, and type. Type is DT_DATETIME, FT_INT, or FT_CHAR.
2. For each column, it searches for a JAM variable destination. If a destination exists, JAM/DB*i* records the length of the variable. If no JAM destination exists for a column, or the destination is an LDB constant, JAM/DB*i* does no fetches for the column. The discussion of JAM destinations is in Section 9.1.1. on page 78.
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. See Section 9.1.2. on page 83.
4. Finally, JAM/DB*i* formats data before writing it to the destination variables if the database column has a date datatype, or if the destination variable has a null, currency, or precision edit. See Section 9.1.3. on page 89.

The sequence above describes a SELECT that writes database column values to individual occurrences of a field, JPL variable, or LDB variable. Developers may also direct the results of a SELECT to two other types of targets. See Section 9.1.4. on page 92 for more information.

9.1.1.

JAM Targets for a SELECT

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 targets with database columns.

- The developer gives a JAM target variable the same name as a database column. This is called *automatic mapping*.
- The developer explicitly declares a JAM variable as the target of a database column. This is called *aliasing*.

Automatic Mapping

By default when executing a SELECT statement, JAM/DB*i* will search for JAM variables with the same names as the specified columns. For the statement,

```
sql SELECT lastname, ssnumber, dept, date FROM emp
```

to return values to JAM variables, the table emp must have at least four columns: lastname, ssnumber, dept, and date. If any of these columns does not exist in the table emp, the engine returns an error.

The application may have a JAM destination variable for none, some, or every named column in the SELECT statement. To return the values of all four columns to the application, then there must be a JAM variable for each column. The variables may be named lastname, ssnumber, dept, and date. If one of these fields does not exist, JAM/DB*i* ignores the values belonging to that particular column.

Developers may also use one or more qualified column names in SELECT statements. For example,

```
sql SELECT emp.lastname, emp.ssnumber, emp.dept, \
   emp.date FROM emp
```

The JAM targets, however, must be given unqualified names: lastname, ssnumber, dept, and date.

JAM/DB*i* also permits the use of the shortcut SELECT statement,

```
sql SELECT * FROM emp
```

Using automatic mapping, JAM/DB*i* looks for a JAM variable for each column in the table emp. Columns without matching variables are simply ignored. This is not treated as an error.

When using automatic mapping, the case of the JAM variable names should correspond to the case flag used in the engine initialization in dbiinit.c. If the engine's case flag is DM_FORCE_TO_LOWER_CASE, the JAM variables for a SELECT 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 use the exact case of the database columns.

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 an aggregate function (COUNT, SUM, AVG, MAX, MIN)

Aliasing is not limited to these conditions. Any or all columns may be aliased if desired. Occasionally, developers like to alias a column if its name is not descriptive or because they wish to name target variables for a particular table and column.

Developers use the command DBMS ALIAS to specify aliases. On some engines, developers may 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's executed with the default cursor. The syntax for assigning aliases to a cursor is either of the following:

```
dbms [WITH CURSOR cursor] ALIAS column1 jam_var1 \
    [, column2 jam_var2 ...]
```

to alias a column name to a JAM variable, or

```
dbms [WITH CURSOR cursor] ALIAS [jam_var1] \
    [, [jam_var2] ...]
```

to alias a column position to a JAM variable. Either named or positional aliasing may be used, but both forms may not be used in a single statement.

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 in `dbiinit.c`. 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.

If an application aliases a column to a JAM variable that does not exist JAM/DB*i* ignores the column's values. This is NOT treated as an error.

Using DBMS ALIAS to Alias Column Names

First consider an example that aliases column names to JAM variables. For example,

```
dbms ALIAS first firstname, last lastname
sql SELECT ssn, last, first FROM emp
```

JAM/DB*i* writes the values from the column `first` to the variable `firstname` and it writes the values of column `last` to the variable `lastname`. Since no alias was given for `ssn`, it maps it to a variable of the same name. See the figure below.

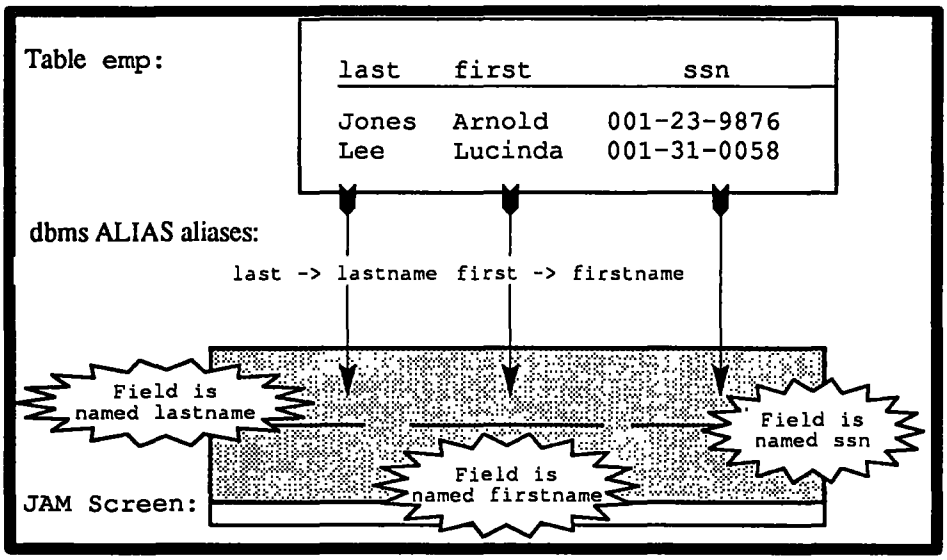


Figure 29: The mapping of `SELECT ssn, last, first FROM emp` when aliases are used.

Aliases may also be given after declaring a named cursor. For example,

```
dbms DECLARE sales_cursor CURSOR FOR \
   SELECT inv#, sale_date, ship_date, amount FROM acc
dbms WITH CURSOR acc_cursor ALIAS "inv#" invoice_id
dbms WITH CURSOR acc_cursor EXECUTE
```

Since `inv#` 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 `inv#` to variable `invoice_id`. 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/DB*i* parses it correctly.

Using `DBMS ALIAS` to Alias Column Positions

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

```
dbms ALIAS min_salary, max_salary, avg_salary
sql SELECT MIN(sal), MAX(sal), AVG(sal) FROM acc
```

JAM/DB*i* writes the aggregate function values to the alias variables. `MIN(sal)` is written to the variable `min_salary`, `MAX(sal)` is written to the variable `max_salary`, and `AVG(sal)` is written to the variable `avg_salary`. Note that there is no automatic mapping available. 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, JAM/DB*i* will attempt to write the first three columns' value to the three positional alias variables. If those variables are no longer available, JAM/DB*i* will ignore the first three columns in the SELECT set.

Using the Engine's SELECT Syntax

Many engines support aliasing in their SELECT 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 may support another syntax. See your database documentation for details.

If an engine supports aliasing in a SELECT statement, JAM/DB*i* will also support it. Developers may 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/DB*i* application,

```
sql SELECT id product_no, supplier, ucost price FROM inv
```

When this statement is executed, the DBMS tells JAM/DB*i* that the columns `product_no`, `supplier`, and `ucost` were selected. JAM/DB*i* will look for variables with those names. If there is a variable `id` available, this SELECT statement will not write to it because the engine has aliased it to `product_no`.

Although this form is supported, we recommend the use of DBMS ALIAS, especially for applications accessing more than one engine. JAM/DB*i* provides identical support for DBMS ALIAS on all engines.

9.1.2.

Number of Rows Fetched

A SELECT set often contains more than one row. JAM/DB*i* must determine how many rows it may fetch at one time from a SELECT set. The rest of the SELECT is fetched by executing one or more DBMS CONTINUE's.

- If an occurrence number was specified with a target variable name, only one row is fetched.
- If a target is a word wrapped array, only one row is fetched.
- If using browse mode, only one row is fetched. (See the engine-specific *Notes*).

Otherwise, JAM/DB*i* 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/DB*i* fetches a row for each occurrence. If the targets do not have the same number of occurrences, JAM/DB*i* finds the target variable with the least number of occurrences and fetches that number of rows. 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,

```
sql SELECT * FROM table
```

The application has onscreen fields 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, the SELECT will fetch only five rows 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 JAM/DB*i* developers must create applications capable of handling a fluctuating number of data rows. Based on the type of data selected and the hardware in use, a developer may use either or both types of scrolling—JAM scrolling or JAM/DB*i* scrolling.

With JAM scrolling, the application uses large scrolling arrays as the destination variables of a SELECT statement. The entire SELECT set is fetched in a single step and the user presses the page up and page down keys (logical keys SPGU and SPGD) to view the rows.

With JAM/DB*i* scrolling, 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 function keys to execute the JAM/DB*i* scrolling commands.

The two methods are described in detail below.

JAM-based Scrolling

JAM-based scrolling is useful for small to mid-sized SELECT sets. The upper limit on the number of rows is 9999, the maximum number of occurrences allowed for a JAM variable. Since the application must keep the entire SELECT set in memory, the realistic limit may be much lower on a platform like MS-DOS or for a SELECT involving many columns.

With this approach, the developer creates large scrolling arrays with more occurrences than the number of rows he or she expects 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 SELECTset. Most often the application examines the value of the global variable @dmretcode. JAM/DB*i* 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 may be examined after a SELECT. See page 93 for more information on these variables. An example procedure is shown below:

```
proc select_all
# DM_NO_MORE_ROWS is an LDB constant.
sql SELECT inv_no, prod_no, prod_desc, quantity, \
   unit_price, total FROM new_sales
if @dmretcode == DM_NO_MORE_ROWS
   msg esmg "All rows returned."
else
   msg emsg "Application could not display all orders."
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, the developer should consider JAM/DB*i*-based scrolling or some other alternative scroll driver.

JAM/DB*i*-based Scrolling

JAM/DB*i*-based scrolling is useful for mid-sized to large SELECT sets. Neither JAM nor JAM/DB*i* impose any limit on the number of rows that may be displayed with this method.[5]

With this approach, developers create non-scrolling arrays. The target fields contain elements to display one or more rows on the screen at time. At least two procedures are needed to view the SELECT set. The first procedure executes the SELECT 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 may be executed many times before the user sees all the rows.

For example, the current screen has fields named for the columns in the table emp. Each field has five elements. The application uses the procedures like the following to select data from a table:

```
proc select_emp
sql SELECT * FROM emp
return

proc continue_select
dbms CONTINUE
return
```

as well as control strings like the following:

```
PF1    ^jpl select_emp
PF2    ^jpl continue_select
```

Assume that table emp contains 12 rows. When the user presses the PF1 key, the application executes the JPL procedure `select_emp` and writes rows 1 through 5 to the screen. If the user presses PF2, the application executes the procedure `continue_select` which clears the arrays and writes rows 6 through 10 to the screen. If the user presses PF2 again, the application executes `continue_select` again which clears the arrays and writes rows 11 and 12 to the screen. If the user presses PF2 a third time, the application does nothing because there are no more rows in the SELECT set.

An application may simulate scrolling through a SELECT set by using the following commands:

5. In multi-user environments developers should know how the engine ensures read consistency: the guarantee that data seen by a SELECT does not change during statement execution. The engine may 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. See the engine-specific *Notes* for more information.

- `DBMS CONTINUE_UP` to scroll up a screenful of rows
- `DBMS CONTINUE_TOP` to scroll to the first screenful of rows
- `DBMS CONTINUE_BOTTOM` to scroll to the last screenful of rows

Some engines have native support for these commands. For example, the engine may buffer the rows in memory on the server. JAM/DB*i* also provides its own support for these commands. Applications may use `DBMS STORE FILE` to set up a continuation file for a named or default `SELECT` cursor. When it is used, JAM/DB*i* buffers `SELECT` rows in a temporary binary file. The syntax of the command is

```
dbms [WITH CURSOR cursor] STORE FILE [file]
```

The command is supported on all engines. To select and view data, an application uses procedures like the following:

```
proc select_emp
dbms STORE FILE
sql SELECT * FROM emp
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
```

as well as control strings like the following:

```
PF1    ^jpl select_emp
PF2    ^jpl scroll_down
PF3    ^jpl scroll_up
PF4    ^jpl scroll_top
PF5    ^jpl scroll_end
```

Using the same number of rows and occurrences as earlier, when the user presses the `PF1` key, the application executes the JPL procedure `select_emp` and writes rows 1 through

5 to the screen. If the user presses PF2, the application executes the procedure scroll_down which clears the arrays and writes rows 6 through 10 to the screen. If the user presses PF3, the application executes scroll_up which clears the arrays and writes rows 1 through 5 to the screen. If the user presses 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.

Although function keys are needed to call the JPL procedures which execute the JAM/DB*i* scrolling commands, end users usually prefer the standard page up and page down keys to the PF keys. The logical keys SPGU and SPGD are not listed in the JAM Control String window of the screen editor but their logical values may be reassigned with the JAM library function sm_keyoption. Therefore, the application may 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 SPGU acts like the function key that calls the scroll up procedure, and calls sm_keyoption so that SPGD acts like the function key that calls the scroll down procedure. The exit function calls sm_keyoption to restore the default behavior.

Developers who wish to use JPL to call sm_keyoption must install the function in the prototyped list in funclist.c. The JPL procedure must also use the decimal or hexadecimal values of the logical keys. The hexadecimal values are listed in the JAM *Configuration Guide* in the key file chapter. An example function is shown below. This function could be used as the field entry and exit on each target field.

```
vars ENTRY(4) EXIT(4)
vars SPGU(6) SPGD(6) APP1(6) APP2(6) KEY_XLATE(1)
cat ENTRY      '128'
cat EXIT       '16'
cat SPGU       '0x113'
cat SPGD       '0x114'
cat APP1       '0x6102'
cat APP2       '0x6202'
cat KEY_XLATE  '2'

proc entry_exit
parms f_no f_data f_occ f_flag
# APP1   ^jpl scroll_up
# APP2   ^jpl scroll_down
   if (f_flag & ENTRY)
   {
      call sm_keyoption :SPGU :KEY_XLATE :APP1
      call sm_keyoption :SPGD :KEY_XLATE :APP2
   }
   else if (f_flag & EXIT)
```

```
{
   call sm_keyoption :SPGU :KEY_XLATE :SPGU
   call sm_keyoption :SPGD :KEY_XLATE :SPGD
}
return
```

JAM/DB*i*-scrolling uses less memory than JAM scrolling. 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 by the developer. 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.

Controlling the Number of Rows Fetched

Developers using field or LDB arrays as the destinations of a SELECT may 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] OCCUR int          [MAX int]
dbms [WITH CURSOR cursor] OCCUR CURRENT      [MAX int]
```

See the *Reference Guide* in this document for information.

Choosing a Starting Row in the SELECT Set

A developer may also change the number of rows fetched by using the command

```
dbms [WITH CURSOR cursor] START int
```

The command tells JAM/DB*i* to read and discard *int* – 1 rows before writing the rest of the SELECT set to JAM variables.

See the *Reference Guide* in this document for information.

9.1.3.

Format of SELECT Results

Before writing a database column value to a JAM variable occurrence, JAM/DB*i* determines the data type of the database column. In all cases, if the value equals the engine's null (e.g., NULL), JAM/DB*i* writes clears the variable. If the variable has a null field edit, JAM automatically converts the null string to the one assigned by the field edit.

If any value is longer than the variable, the data is truncated.

Character Column

If a column has a character datatype, the value is simply written to the target variable. If the variable has a word wrap edit or a right-justified edit, the edit is applied.

Date-time Column

If a column has a date datatype, JAM/DB*i* formats the value before writing it to a JAM variable. If the variable has a date-time edit, JAM/DB*i* uses it. If the variable does not, JAM/DB*i* 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, 1991 10:05:03` would be formatted as `4/1/91 10:05`. When the message file default is used, JAM/DB*i* assumes a 12-hour clock.

See the *Author's Guide* and the *Configuration Guide* in the JAM documentation for information on date-time formats.

Numeric Column

If a column has an integral type, JAM/DB*i* 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 field.

If a column has a real type, JAM/DB*i* converts the value to a double. Before writing the value to a JAM variable, JAM/DB*i* determines the precision by examining the variable's currency and/or C type edit.

- *The field has a currency edit, but no C type edit.* 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.

- *The field has a C type edit, but no currency edit.* 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.
- *The field has a currency edit and C type edit that 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/DB*i* 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. If it uses the C type precision, it adjusts by standard rounding to the precision.
- *The field has neither a currency edit or a C type edit.* The precision defaults to 2.

See the *Author's Guide* in the JAM documentation for more information on currency edits.

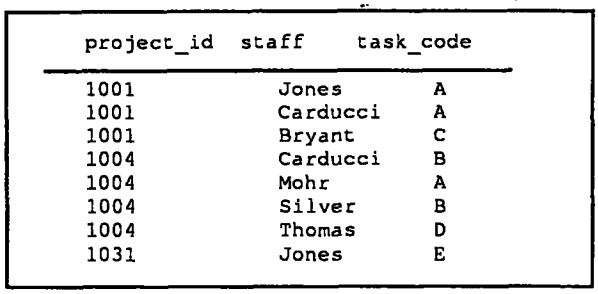
Fetching Unique Column Values

By default, when a column is selected JAM/DB*i* returns all values. JAM/DB*i* also provides a command for displaying only a column's unique values,

```
dbms [WITH CURSOR cursor] UNIQUE column [column ...]
```

JAM/DB*i* replaces a repeating value with the 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.

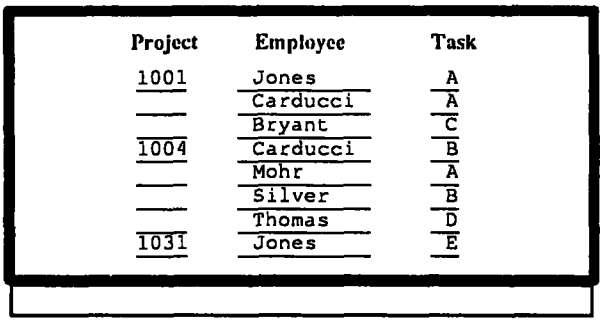


project_id	staff	task_code
1001	Jones	A
1001	Carducci	A
1001	Bryant	C
1004	Carducci	B
1004	Mohr	A
1004	Silver	B
1004	Thomas	D
1031	Jones	E

Figure 30: The primary key of table `projects` is (`project_id, staff`).

```
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
```

Below 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 31: The JAM layout is easier to read than the table layout.

See the *Reference Guide* in this document for more information.

9.1.4.

Redirecting SELECT Results to Other Targets

Occasionally, developers need other destinations for SELECT statements. JAM/DB*i* provides a feature for concatenating a full result row and writing it to either a JAM variable or a text file.

```
dbms [WITH CURSOR cursor] CATQUERY TO jam_var \
    [SEPARATOR text] [HEADING [ON | OFF] ]

dbms [WITH CURSOR cursor] CATQUERY TO FILE filename \
    [SEPARATOR text] [HEADING [ON | OFF] ]
```

JAM/DB*i* also provides a command for formatting the results,

```
dbms [WITH CURSOR cursor] FORMAT [column] format
```

See the *Reference Guide* in this document for details.

9.2. STATUS AND ERROR CODES

JAM/DB*i* supplies several pre-defined variables where it stores error and status data for the application. These variables are

- `@dmretcode` The status of the last executed `dbms` or `sql` 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` or `sql` statement. Its value is 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` or `sql` 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` or `sql` statement. If `@dmengerrcode` is empty, this variable is also empty.
- `@dmengwarncode` An engine-specific warning code or bit setting for the last executed `dbms` or `sql` 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` or `sql` statement. If `@dmengwarn` 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. See the engine-specific *Notes*.
- `@dmserial` An engine-generated value for a serial column. Its value is 0 or an appropriate serial value for the column. See the engine-specific *Notes*.

After executing a statement JAM/DB*i* updates these variables with any error, warning, or status information returned by the engine. In addition to the engine-specific codes and messages, JAM/DB*i* also supplies engine-independent codes and messages to the variables `@dmretcode` and `@dmretmsg`.

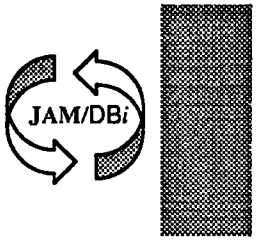
These global variables are available throughout the application from both JPL and C. Note that JAM/DB*i* does not automatically display these values, except in the case of error messages.

JAM/DB*i* uses a default error handler when executing `dbms` and `sql` commands from JPL or C. If a JAM/DB*i* error occurs, the default error handler displays an error message. The source of the message depends on the message flag used to initialize the engine, either the `DM_DEF_ENG_MSG` flag or the `DM_DEF_DBI_MSG` flag.

If a JAM/DB*i* error occurs while executing JPL, the default error handler displays a message and JAM displays the `dbms` or `sql` statement where the error occurred. When the last message is acknowledged, JAM/DB*i* aborts the JPL procedure where the error occurred. An aborted JPL procedure always returns –1 to its caller.

If a JAM/DB*i* error occurs while executing one of the C library functions, the default error handler displays the error message and JAM returns –1 to the function.

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 display these values to the enduser. This topic is covered in the next chapter.



Chapter 10.
Hook Functions

JAM/DB*i* provides three hooks for developer-written functions. They are the following

- `ONENTRY` This function is called before executing any `dbms` or `sql` command from JPL or C.
- `ONEXIT` This function is called after executing any `dbms` or `sql` command from JPL or C.
- `ONERROR` This function is called if an error occurs while executing any `dbms` or `sql` command from JPL or C.

JAM/DB*i* hook functions may be written in JPL or C.

A JPL hook function is installed like the following:

```
dbms ONXXXX 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. See the JPL Guide for more information.

A C hook function is installed like the following:

```
dbms ONXXXX CALL function
```

where ***function*** is a prototyped function. A prototyped function appears on JAM's `PROTO_FUNC` list. As a JAM/DB*i* hook function, 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 },
}
```

Please consult the JAM *Programmer's Guide* for more information on prototyped functions.

10.1. ONENTRY FUNCTION

Before executing a `dbms` or `sql` command from JPL or C, JAM/DB*i* will execute 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 remap cursor control keys or change protection edits on fields.

To install an ONENTRY function, use one of the following:

```
dbms ONENTRY JPL entry_point
dbms ONENTRY CALL function
```

To turn off the ONENTRY function, execute the command with no arguments:

```
dbms ONENTRY
```

10.1.1. ONENTRY Function Arguments

An ONENTRY hook function receives 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 `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 this value is 0.

10.1.2. ONENTRY Return Codes

In the present release, 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 returns 0.

10.1.3.
Example ONENTRY Functions

The following sample function logs the current statement in a text file.

```
/* This function is installed as a prototyped function.*/
/* It writes the current time, name of the current */
/* engine, and the command which JAM/DBi will execute */
/* to a file called dbi.log. */

/* dbms ONENTRY CALL dbientry       */

#include "smdefs.h"

int
dbientry (stmt, engine, flag)
char *stmt;
char *engine;
int flag;
{
   FILE *fp;
   time_t timeval;

   fp = fopen ("dbi.log", "a");
   timeval = time(NULL)
   fprintf (fp, "%s\n%s\n%s\n\n",
            ctime(&timeval), engine, stmt);
   fclose (fp);
   return 0;
}
```

This sample function displays a message before performing any JAM/DB*i* operations.

```
# dbms ONENTRY JPL entrymsg

proc entrymsg
   msg setbkstat "Processing. Please be patient..."
   flush
   return 0
```

10.2. ONEXIT FUNCTION

After executing a dbms or sql command from JPL or C, JAM/DB*i* will execute the application's installed ONEXIT function. An ONEXIT function is useful for logging or debugging statements. You may also use an ONENTRY function to modify the JAM environment, for instance remap cursor control keys or change protection edits on fields. This function is useful for checking error and status codes after each command.

10.2.1. ONEXIT Function Arguments

An ONEXIT hook function receives 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 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 ONEXIT function its value is 1.

10.2.2. ONEXIT Return Codes

The return code from an ONEXIT function is ignored unless an error occurred while executing a sql or dbms command using JPL. If the return code from the function is non-zero, JAM/DB*i* 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.

10.2.3. Example ONEXIT Function

This sample function looks for the no more rows codes after executing a command.

```
# dbms ONEXIT JPL checkstat

# DM_NO_MORE_ROWS is an LDB constant set to 53256

proc checkstat
parms stmt engine flag
if @dmretcode != 0
{
   if @dmretcode == DM_NO_MORE_ROWS
   {
            msg emsg "All rows were returned."
            return 0
   }
   msg emsg "Error executing " stmt "%N" \
            @dmretmsg "%N" @dmengerrrmsg
   return 1
}
return 0
```

10.3.
ONERROR FUNCTION

If a JAM/DB*i* error occurs while executing a dbms or sql command from JPL or C, JAM/DB*i* will execute the application's installed ONERROR function. An ONEXIT function usually displays the values of the global error variables @dmretmsg and @dmengerrmsg. It may also display the text of the command that failed. The application may use this function to log error information in a text file.

There are two classes of JAM/DB*i* errors:

- *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 a keyword, executing a cursor that has not been declared, or failing to declare a connection before executing an sql statement. These errors are detected by JAM/DB*i* and reported using standard JAM/DB*i* error codes and messages. 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 reported by the JAM/DB*i* interface. These errors update the global variables `@dmretcode`, `@dmretmsg`, `@dmengerrcode`, `@dmengerrmsg`.

Note that JAM and JPL errors are not a class of JAM/DB*i* errors. In addition to a JAM/DB*i* error, 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 developers may need special handling for these errors is during transaction processing. This is discussed in Chapter 11.

An ONERROR function overrides JAM/DB*i*'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/DB*i* calls the ONEXIT function, then the ONERROR function.

To install an ONERROR function, use one of the following:

```
dbms ONERROR JPL entry_point
dbms ONERROR CALL function
```

To turn off the ONERROR function and reinstall the default error handler, execute the command with no arguments:

```
dbms ONERROR
```

10.3.1.

ONERROR Function Arguments

An ONERROR hook function receives 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 `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 ONERROR function its value is 2.

10.3.2.

ONERROR Return Codes

If an application is using an installed error handler, the error handler determines the handling for JAM/DB*i* errors that occur while using JPL.

If a JAM/DB*i* error occurs 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 however the JPL procedure resumes control; JAM will execute the next statement in the JPL procedure.

If a JAM/DB*i* error occurs while executing C, 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.

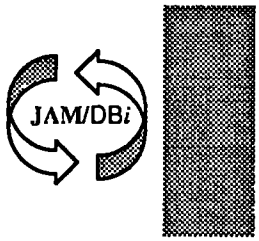
10.3.3.

Example ONERROR Function

```
# DM_ALREADY_ON is an LDB constant.

proc dbi_error_handler
parms stmt engine flag

   if (@dmretcode == DM_ALREADY_ON)
   {
      msg emsg "You are already logged on."
      return 0
   }

   if (@dmengerrcode != 0)
   {
         msg emsg @dmretmsg
         jpl engine_errors :engine
   }
   else
   {
```

```
        msg emsg "Application Error:  " \
           @dmretmsg \
           "See the DBA for assistance."
     }

     return 1

  proc engine_errors
  parms engine_name
     if engine_name == "xyzdb"
     ...
  # Examine DBMS ERROR codes here.
  ...
```

This procedure first checks if the 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 calls another subroutine to check for engine-specific errors. For any other errors, it displays the standard JAM/DB*i* message.



Chapter 11.
Transactions

In addition to the data access capabilities of an engine, JAM/DB*i* supports the engine's transaction processing capabilities.

A 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. That is, the update takes the database from one consistent state to another. Using the familiar personnel database described throughout the document, consider these possible transactions:

- *An employee review transaction.* It involves: an insert to the table `review` supplying a social security number, review date, new salary, and new grade level and an update to the employee's current salary in the table `acc`.
- *A new employee transaction.* It involves: an insert to the table `emp` supplying the employee's social security number, name, and home address; an insert to the table `review` supplying the employee's social security number, hire date, salary, and grade; and an insert to the table `acc` supplying the employee's social security number, current salary, and number of tax exemptions.

Transaction processing is sometimes a difficult topic for new developers. For one, 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/DB*i* 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.

11.1. 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/DB*i* 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 COMMI 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 once the transaction is completed, not when 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 interfering with one another
- produces applications which are less database-dependent

Finally, although vendors supply commands for transaction processing in their SQL language, developers should use those provided by JAM/DB*i* either with the JPL command `dbms` or the library routine `dm_dbms`. Using `sql` or `dm_sql` to handle transaction processing like commit and rollback is NOT recommended. Using the DBMS versions permits JAM/DB*i* to establish necessary structures and it provides better error handling if a transaction fails.

11.2.
ERROR PROCESSING FOR A TRANSACTION

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, JAM/DB*i* must execute a rollback when the engine returns an error to the application. For example, the engine might reject 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, developers very likely want to execute a rollback for any error that occurs during the transaction, including an error detected by JAM or JAM/DB*i* before a statement is passed to the engine. An error detected by JAM or JAM/DB*i* 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 JAM/DB*i* C library functions, JAM and JAM/DB*i* error codes 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/DB*i* library functions rather than JPL.

If the transaction processing is done in JPL with `dbms`, developers should use the JPL command `retvar` to declare a return variable. A `retvar` variable is set to 0 if a called procedure returns 0 (the default for success) or if a `dbms` or `sql` statement executes without error. If a called procedure aborts because of a JAM error, a `retvar` variable is set to –1. If an installed error handler is called, a `retvar` variable is set to the handler's return code. The JPL Guide in Volume II of the JAM manual has a complete description of this command. The examples in this chapter use `retvar` so that a transaction is rolled back for all JAM/DB*i* and JAM errors.

The best method for transaction processing in JPL uses a generic JPL procedure as a transaction hander. This procedures does the following:

- defines and declares a JPL return variable, ***jpl_retcode***.
- calls a JPL subroutine that contains the actual transaction statements.

- on return from the subroutine, examines the JPL return 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 may 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
{
    parms subroutine
    vars jpl_retcode
    retvar jpl_retcode
# Call the subroutine.
        jpl :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
}

proc update_emp
{
    ....
    dbms COMMIT
    return 0
}
```

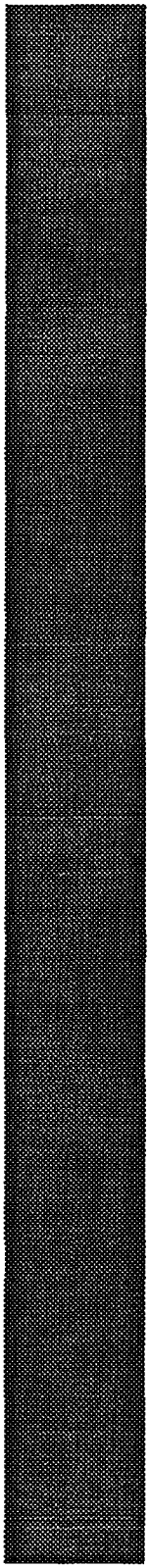
To execute the update transaction, the application should execute

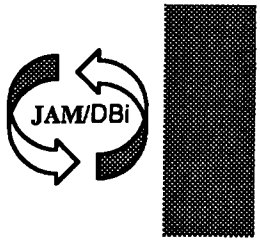
```
jpl tran_handle update_emp
```

Once `tran_handle` has set up the return variable, it calls the procedure `update_emp`. Whether `update_emp` is successful or unsuccessful, control is always returned to `tran_handle`.

In the engine-specific *Notes*, there is a list and description of the supported transaction commands with more examples.

JAM/DB*i*
Reference Guide





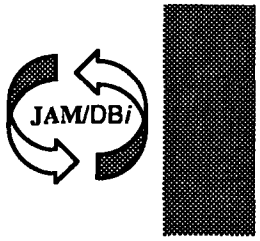
Chapter 12.
JAM/DBi *Reference Overview*

This guide has a reference chapter on each of the following:

- JAM/DBi global variables
- DBMS commands
- JAM/DBi library functions
- JAM/DBi utilities

Each reference chapter provides a summary of the topic, and a reference page for each command, function, or utility. The reference pages use following notation:

`literal`	This font indicates text that the developer will type verbatim. In particular, it is used for all examples and for the names of JAM library functions, JPL commands, or utilities.
SMALL CAPS	Uppercase is used for SQL keywords and dbms command keywords. This use of case is stylistic. Case is significant only for identifiers—names of fields, columns, tables, variables. functions, etc.
italics	Bold italics show where variable or procedure names should appear. Text in this font should be replaced with a value appropriate for the application.
[*x*]	Brackets indicate an optional element. The brackets should not be typed.
{*x* | *x*}	Braces indicate a series of valid options. At least one option must be used. The braces should not be typed.
x...	Ellipses indicate that an element may be repeated one or more times.



Chapter 13.
DBMS Global Variables

This chapter summarizes and categorizes the JAM/DB*i* global variables.

13.1.
VARIABLE OVERVIEW

The global JAM/DB*i* variables are automatically defined by JAM/DB*i* at initialization. All JAM/DB*i* global names 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 `sm_n_getfield` and `sm_n_fptr`.

The variables are automatically maintained by JAM/DB*i*. Before executing a `dbms` or `sql` statement, JAM/DB*i* clears the contents of all its global variables. After executing the statement and before returning control to the application, JAM/DB*i* updates the variables to indicate the current status.

13.1.1.
Error Data

`@dmretcode`	JAM/DB*i* error code. Codes are the same for all engines.
`@dmretmsg`	JAM/DB*i* error message. Messages are the same for all engines.
`@dmengerrcode`	Engine error code. Codes are unique to the engine.

`@dmengerrmsg`	Engine error message. Messages are unique to the engine. Some engines do not supply messages.

13.1.2. Status Data

`@dmretcode`	JAM/DB*i* status code for "no more rows" or "end of proc."
`@dmretmsg`	JAM/DB*i* status message for "no more rows" or "end of proc."
`@dmengreturn`	Engine return code from a stored procedure. Not used by all engines.
`@dmrowcount`	Count of the number of rows fetched to JAM by the last `SELECT` or `CONTINUE`. Used by all engines.
`@dmserial`	A serial value returned after inserting a row into a table with a serial column. Not used by all engines.
`@dmengwarncode`	A code or byte signalling a non-fatal error or unusual condition. Used by all engines.
`@dmengwarnmsg`	A message corresponding to an engine warning code. Not used by all engines.

13.2. VARIABLE REFERENCE

The rest of this chapter contains a reference page for each global variable. Since some variables store engine-specific values, additional information is provided in the engine-specific *Notes*.

Each reference page has the following sections:

- A description of the variable.
- A list of related variables and commands.
- An example.

The variables are documented in alphabetical order.

@dmengerrcode

contains an engine-specific error code

DESCRIPTION

JAM/DB*i* sets this variable to 0 before executing a `dbms` or `sql` statement. If the engine detects an error, JAM/DB*i* writes the engine's error code to `@dmengerrcode`.

Note that a 0 value in this variable does not guarantee that the last statement executed without error. Some errors are detected by JAM/DB*i* before a request is made to the engine. For example, if an application attempts a `SELECT` before declaring a connection, JAM/DB*i* detects the error. Use the global variable `@dmretcode` to check for JAM/DB*i* errors.

Because the value of `@dmengerrcode` is engine-specific, the use of an installed error handler is strongly recommended. The application may test for engine-specific errors within the error handler or in a multi-engine application, the error handler may call another function to do this.

Please consult the engine-specific *Notes* for more information about the codes for your engine.

SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

RELATED FUNCTIONS

`dbms ONERROR [JPL` ***entrypoint*** `| CALL` ***function***`]`

RELATED VARIABLES

`@dmengerrmsg`

`@dmretcode`

`@dmretmsg`

EXAMPLE

```
proc dbi_errhandle
parms stmt engine flag
if @dmengerrcode == 0
   msg emsg @dmretmsg
else if engine == "xyzdb"
   jpl xyzerror @dmengerrcode
else if engine == "oracle"
```

```
   jpl oraerror @dmengerrcode
else
   msg emsg "Unknown engine."
return 1

proc xyzerror
# Check for specific xyzdb error codes.
parms error
   if error == 90931
      msg emsg "Invalid user name."
   else if error == ...
   ...
   else
      msg emsg @dmengerrmsg
   return
```

@dmengerrmsg

contains an engine-specific error message

DESCRIPTION

JAM/DB*i* clears this variable before executing a new `@dbms` or `@sql` statement. If the engine returns an error message after attempting to execute the statement, JAM/DB*i* writes the message to this variable.

If `@dmengerrcode` is 0, this variable contains no message.

Please consult the engine-specific *Notes* for more information about the error messages for your engine.

SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

RELATED FUNCTIONS

`dbms ONERROR [JPL` ***entrypoint*** `| CALL` ***function***`]`

RELATED VARIABLES

`@dmengerrcode`

`@dmretcode`

`@dmretmsg`

EXAMPLE

```
proc dbi_errhandle
parms stmt engine flag
if @dmengerrcode == 0
   msg emsg @dmretmsg
else
   msg emsg @dmretmsg "%N" @dmengerrmsg
return 1
```

@dmengreturn

contains a return code from a stored procedure

DESCRIPTION

If your engine supports stored procedures and stored procedure return codes, use this variable to get a procedure's return or status code.

By default, JAM/DB*i* will pause the execution of a stored procedure if the procedure executes a SELECT statement and the number of rows in the SELECT set is greater than the number of occurrences in the JAM destination variables. The application must execute DBMS CONTINUE or DBMS NEXT to resume execution. If the value of `@dmengreturn` is null after calling a stored procedure, the procedure may be pending. If the engine has completed the execution of the procedure, `@dmretcode` will contain the DM_END_OF_PROC code and `@dmengreturn` will contain the procedure's return code.

Note that the value of this variable will be cleared once another `dbms` or `sql` statement is executed. If the application needs this value for a longer period of time, it should copy it to a standard JAM variable or some other static location.

SEE ALSO

Notes

RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] NEXT

dbms [WITH CURSOR cursor] SET \
   [SINGLE_STEP|STOP_AT_FETCH|EXECUTE_ALL]
```

RELATED VARIABLES

```
@dmretcode

@dmretmsg
```

EXAMPLE

```
# create proc checkid @id char(15) as
# declare @idcount int
# select @idcount = SELECT COUNT (*) FROM products WHERE
# id = @id
# if @idcount == 1
#     return 1
```

```
# else
#     return -1


sql  EXEC checkid :+id
if @dmengreturn == 1
   jpl addrow
else if @dmengreturn == -1
   msg emsg "Sorry, " id " is not a valid code."
return
```

@dmengwarncode

contains an engine-specific warning code

DESCRIPTION

Most engines supply a mechanism for signalling an unusual, but non-fatal condition.

Some engines use an eight-element array. If there is a warning, it sets the first element to indicate a warning and then sets one or more additional elements to describe the warning. Other engines uses codes and messages similiar to those it uses for errors. Those of a high severity are handled as errors and those of a low severity are handled as warnings. Please consult the engine-specific *Notes* for information about your engine and for an example.

By default, JAM/DB*i* ignores warnings. If an application needs to alert users to warning codes, it must use a JPL or C function to check for them. There is no default warning handler. The most efficient way to process warning codes is with an installed exit handler.

SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

RELATED FUNCTIONS

`dbms ONEXIT [JPL` ***entrypoint*** `| CALL` ***function***`]`

RELATED VARIABLES

`@dmengwarnmsg`

@dmengwarnmsg

contains an engine-specific warning message

DESCRIPTION

Most engines supply a mechanism for signalling an unusual, but non-fatal condition. Some engines uses a warning array or byte. These engines do not supply warning messages and therefore do not use `@dmengwarmsg`. Others use a code and message for low-severity errors. Please consult the engine-specific *Notes* for information about your engine and for an example.

By default, JAM/DB*i* ignores warnings. If an application needs to alert users to warning codes or messages, it must use a JPL or C function to check for them. There is no default warning handler. The most efficient way to process warning values is with an installed exit handler.

SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

RELATED FUNCTIONS

`dbms ONEXIT [JPL` ***entrypoint*** `| CALL` ***function***`]`

RELATED VARIABLES

`@dmengwarncode`

@dmretcode

contains an engine-independent error or status code

DESCRIPTION

Before executing a new `dbms` or `sql` statement, JAM/DB*i* writes a 0 to `@dmretcode`. If the statement fails because of a JAM/DB*i* or engine error, JAM/DB*i* writes an error code to `@dmretcode` describing the failure. These codes are defined in `dmerror.h` and are engine-independent. The codes are 5-digits long. See *Appendix* B. for a listing.

Usually a non-zero value in `@dmretcode` indicates that an error occurred. The default or an installed error handler is called for an error. If the default handler is in use, JAM/DB*i* will display an error message. If the application has installed its own error handler, the installed function controls what messages are displayed. Since these codes are generic, applications often need engine-specific error values as well. Engine-specific error codes are written to `@dmengerrcode`.

There are two non–zero codes for `@dmretcode` which are not errors: `DM_NO_MORE_ROWS` and `DM_END_OF_PROC`. When an engine indicates that it has returned all rows for a `SELECT` set, JAM/DB*i* writes the `DM_NO_MORE_ROWS` code to `@dmretcode`. Since this is not considered an error, JAM/DB*i* does not call the default or an installed error handler. You may test for `DM_MORE_ROWS` after executing a `SELECT` or in an exit handler. JAM/DB*i* uses `DM_END_OF_PROC` with engines that support stored procedures. When an engine indicates that it has completed executing the stored procedure, JAM/DB*i* writes the `DM_END_OF_PROC` code to `@dmretcode`. This is not an error. An application may test for this code in an exit procedure or after calling a stored procedure. See the engine-specific *Notes* for information on stored procedures.

SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

Appendix B.

RELATED FUNCTIONS

`dbms ONERROR [JPL` ***entrypoint*** `| CALL` ***function***`]`

`dbms ONEXIT [JPL` ***entrypoint*** `| CALL` ***function***`]`

RELATED VARIABLES

`@dmengerrcode`

`@dmengerrmsg`

@dmretmsg

EXAMPLE

```
# The following are LDB constants.
# DM_ALREADYON = 53251
# DM_LOGON_DENIED = 53253
# DM_NO_MORE_ROWS = 53256


proc dbi_errhandle
parms stmt engine flag
# Check for logon errors.
if @dmretcode == DM_ALREADYON
   return 0
else if @dmretcode == DM_LOGON_DENIED
   msg emsg @dmretmsg "%N" @dmengerrmsg
....
return 1

proc dbi_exithandle
parms stmt engine flag
if @dmretcode == DM_NO_MORE_ROWS
   msg emsg "All rows returned."
return 0
```

@dmretmsg

contains an engine-independent error or status message

DESCRIPTION

Before executing a new `dbms` or `sql` statement, JAM/DB*i* clears `@dmretmsg`. If the statement fails because of a JAM/DB*i* or engine error, JAM/DB*i* writes an error message to `@dmretmsg` describing the failure. These messages are defined in JAM's `msgfile` and are engine-independent. See *Appendix* B. for a listing.

Note that if `@dmretcode` is 0, `@dmretmsg` is always empty.

SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

RELATED FUNCTIONS

`dbms ONERROR [JPL` ***entrypoint*** `| CALL` ***function***`]`

`dbms ONEXIT [JPL` ***entrypoint*** `| CALL` ***function***`]`

RELATED VARIABLES

`@dmengerrcode`

`@dmengerrmsg`

`@dmretcode`

EXAMPLE

```
proc dbi_errhandle
parms stmt engine flag
   msg emsg "Statement " stmt " failed." \
   @dmretmsg "%N" @dmengerrmsg
   return 1
```

@dmrowcount

contains a count of the number of rows fetched to JAM by a SELECT or CONTINUE

DESCRIPTION

Before executing a new `dbms` or `sql` statement, JAM/DB*i* writes a 0 to this variable. If the statement fetches rows, JAM/DB*i* updates the variable writing the number of rows fetched to JAM variables.

Most SQL syntaxes provide an aggregate function `COUNT` to count the number of values in a column or the number of rows in a `SELECT` set. The value of `@dmrowcount` is NOT the number of rows in a `SELECT` set; rather, it is the number of rows returned to JAM variables. Therefore if a `SELECT` set has 14 rows in total, and its target JAM variables are arrays, each with ten occurrences, `@dmrowcount` will equal 10 after the `SELECT` is executed, and 4 after the `DBMS CONTINUE` is executed. If `DBMS CONTINUE` were executed a second time, `@dmrowcount` would equal 0.

The integer written to `@dmrowcount` is either less than or equal to the maximum number of rows that can be written to the targeted JAM destinations; the maximum number of rows is the number of occurrences in a destination variable. If the value in `@dmrowcount` is less than the maximum number of occurrences, then the entire `SELECT` set was written to the target variables and no further processing is needed. If `@dmrowcount` equals the maximum number of occurrences, then the `SELECT` may have fetched more rows than will fit in the variables. To display the rest of the `SELECT` set, the application must execute `DBMS CONTINUE` until `@dmrowcount` is less than the maximum number of occurrences (or equals 0) or until `@dmretcode` receives the `DM_NO_MORE_ROWS` code.

SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

RELATED FUNCTIONS

`dbms ONEXIT [JPL` ***entrypoint*** `| CALL` ***function***`]`

RELATED VARIABLES

`@dmretcode`

EXAMPLE

```
proc get_selection
   sql SELECT * FROM movie_archive WHERE subject=:+subj
   jpl check_count
   return

proc check_count
# If rows are returned but not the NO_MORE_ROWS code,
# let the user know there are rows pending.
   if (@dmrowcount > 0) && (@dmretcode != DM_NO_MORE_ROWS)
      msg setbkstat "Press %KPF1 to see more."
   else
      msg setbkstat "All rows returned."
   return

proc get_more
# This function is called by pressing PF1.
# It retrieves the next set of rows.
   dbms CONTINUE
   jpl check_count
   return
```

@dmserial

contains a serial column value after performing INSERT

DESCRIPTION

Some engines supply the datatype `serial` to assist endusers and applications that need to assign a unique numeric value to each row in a table. When an application inserts a row in a table with a serial column, the engine generates a serial number, inserts the row with the number, and returns the number to the application. See the engine-specific *Notes* for information about support for this on your engine.

Before executing a new `dbms` or `sql` statement, **JAM/DB*i*** writes a 0 to `@dmserial`. If the statement is an `INSERT` and the engine returns a serial value, **JAM/DB*i*** writes the value to `@dmserial`. Since this variable is cleared before executing a new `sql` or `dbms` statement, you must copy its value to another location if you wish to use the value in subsequent statements.

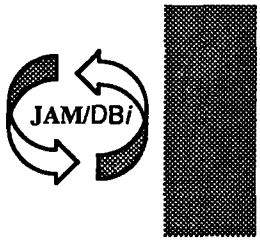
SEE ALSO

JAM/DB*i* *Developer's Guide*, Section 9.2. and Chapter 10..

EXAMPLE

```
# Column order_num is a SERIAL column.

proc new_order
vars i(3) order_id(5)

   dbms BEGIN
   # First INSERT row into invoices table.
   # Column order_id in table invoices is a SERIAL.
      sql INSERT INTO invoices \
         (order_id, order_date, cust_num) VALUES \
         (0, :+today, :+cust_num)

   # Copy the serial value to a JPL variable for use with
   # subsequent INSERTS.
      cat order_id @dmserial

   # Use order number to insert new rows to the orders
   # table. Column order_id in table orders is an INT.
      for i=1 while i<=max step 1
```

```
        sql INSERT INTO orders \
        (order_id, part_id, quant, u_cost) VALUES \
        (:order_id, :+part_id[i], :+quant[i], :+u_cost[i])
    dbms COMMIT

msg emsg "Order completed. Invoice number is " order_num
    return
```



Chapter 14.
DBMS Commands

This chapter summarizes and categorizes the DBMS commands supported by all engines. These commands are executed with the JPL command dbms or the C library function dm_dbms. Commands that are specific to an engine are described in *Notes*. This includes the transaction commands and any special feature commands.

14.1.
DBMS COMMAND OVERVIEW

The DBMS commands fall into several categories. The sections below summarize the commands in each category. Some commands may be listed more than once.

14.1.1.
Engine Selection

ENGINE	set the default engine for the application
WITH ENGINE	set the default engine for the duration of a command

14.1.2.
Using Connections

CLOSE CONNECTION	close a named connection

CLOSE_ALL_CONNECTIONS	close all connections on all engines
CONNECTION	set a default connection and engine for the application
DECLARE CONNECTION	declare a named connection to an engine
WITH CONNECTION	set the default connection for the duration of a command

14.1.3.
Using Cursors

CLOSE CURSOR	close a cursor
CONTINUE	fetch the next screenful of rows from a SELECT set
DECLARE CURSOR	declare a named cursor
EXECUTE	execute a named cursor
WITH CURSOR	specify the cursor to use for a statement

14.1.4.
Changing SELECT Behavior

ALIAS	name a JAM variable as the destination of a selected column or an aggregate function
BINARY	create a JAM/DB*i* variable for fetching binary values
CATQUERY	redirect SELECT results to a file or a JAM variable
FORMAT	format the results of a CATQUERY
OCCUR	set the number of rows for JAM/DB*i* to fetch to an array and choose an occurrence where JAM/DB*i* should begin writing result rows
START	set the first row for JAM/DB*i* to return from a SELECT set
UNIQUE	suppress repeating values in a selected column

14.1.5.
Paging through Multiple Rows

CONTINUE	fetch the next screenful of rows from a SELECT set

CONTINUE_BOTTOM	fetch the last screenful of rows from a SELECT set
CONTINUE_DOWN	fetch the next screenful of rows from a SELECT set
CONTINUE_UP	fetch the previous screenful of rows from a SELECT set
CONTINUE_TOP	fetch the first screenful of rows from a SELECT set
STORE FILE	store the rows of a SELECT set in a temporary file so that the application may scroll through the rows

14.1.6. Handling Binary Data

BINARY	define one or more binary variables

14.1.7. Status and Error Processing

ONENTRY	install a JPL procedure or C function which JAM/DB*i* will call before executing a sql or dbms statement
ONERROR	install a JPL procedure or C function which JAM/DB*i* will call whenever a sql or dbms statement fails
ONEXIT	install a JPL procedure or C function which JAM/DB*i* will call after executing a sql or dbms statement

14.2. COMMAND REFERENCE

The rest of this chapter contains a reference page for each DBMS command. The commands in this chapter may be executed with the JPL command dbms or the library function dm_dbms. Some engines may support additional commands. See the engine-specific *Notes* for a list of such commands.

Each reference page has the following sections:

- A synopsis of the command, including a listing of available keywords and arguments.

- A description of the command.
- A list of related commands.
- An example.

ALIAS

set aliases for a declared or default SELECT cursor

SYNOPSIS

```
dbms [WITH CURSOR cursor] ALIAS [ column jamvar \
   [, column jamvar ...] ]

dbms [WITH CURSOR cursor] ALIAS [ jamvar [, jamvar ...] ]
```

DESCRIPTION

By default, database values are written to JAM variables with the same names as the selected columns. Use this command to map a database column or value to any JAM variable.

If a column name is given, the column is associated with the variable name that follows it. For example,

```
dbms ALIAS lastname emp_lastname, street address
```

If the column `lastname` is selected with the default cursor, JAM/DB*i* will write its values to the JAM variable `emp_lastname`. If the column `street` is selected with the default cursor, JAM/DB*i* will write its values to the JAM variable `address`. For all other columns selected with the default cursor, JAM/DB*i* will write to a variable with the same (unqualified) name as the selected column.

If ***column*** contains characters not permitted in JAM identifiers, enclose ***column*** in quotes to ensure correct parsing.

The case of ***column*** should match the setting of the case flag used to initialize the engine in `dbiinit.c`. For example, if the case flag is `DM_FORCE_TO_LOWER_CASE`, ***column*** must be typed in lower case. The case of ***jamvar*** must be the case used to name the JAM variable. If ***jamvar*** does not exist, JAM/DB*i* ignores the column when it executes the `SELECT`.

If no ***column*** arguments are given, the association is positional. For example,

```
dbms ALIAS emp_var, , abc
```

If the above statement is executed, then each time values are selected with the default cursor, JAM/DB*i* will write the values of the first and third columns to the JAM variables `emp_var` and `abc` respectively. For all other columns selected with the default cursor, JAM/DB*i* will write to a variable with the same (unqualified) name as the selected column. The order of column names in the `SELECT` statement determines the mapping. The case of

jamvar must be the case used to name the JAM variable. If ***jamvar*** does not exist, JAM/DB*i* simply ignores the column when it executes the SELECT.

Named and positional aliases may not be assigned in a single statement.

If aliases are declared for a CATQUERY cursor with the HEADING ON option, JAM/DB*i* will use the aliases rather than the column names to build the heading. The alias for a column selected with a CATQUERY cursor may be enclosed in quotes. This permits a column heading to use embedded spaces. For example,

```
dbms DECLARE emp_cursor CURSOR FOR \
   SELECT first, last, dept FROM emp
dbms WITH CURSOR emp_cursor CATQUERY TO FILE emp_list
dbms WITH CURSOR emp_cursor ALIAS \
   "First Name", "Last Name", Department
dbms WITH CURSOR emp_cursor EXECUTE
```

Aliasing for a cursor is turned off by executing the DBMS ALIAS command with no arguments. Closing a cursor also turns off aliasing. If a cursor is redeclared without being closed, the cursor keeps the aliases. Aliases do not affect INSERT, UPDATE, or DELETE statements.

This command is necessary if the name of a selected column is not a valid JAM variable name, if the application is selecting values from different tables which use the same column name for different values, or if a selection is not a column value, but the value of an aggregate function.

SEE ALSO

JAM/DB*i* *Developer's Guide*, page 79.

RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor ] CATQUERY [TO [FILE] destination \
   [SEPARATOR "text"] [HEADING [ON | OFF]] ]
```

```
[WITH CURSOR cursor]
```

EXAMPLE

```
# Assign aliases by named for a declared cursor.
dbms DECLARE x CURSOR FOR \
   SELECT lname, fname, code FROM directory
dbms WITH CURSOR x ALIAS \
   lname last, fname first, code dept
dbms WITH CURSOR x EXECUTE
dbms WITH CURSOR x ALIAS
```

```
# Set a positional alias for the 2nd and 4th columns.
# Use automatic mapping for the 1st and 3rd columns.
dbms ALIAS , var_x, , var_y
sql SELECT ssn, last, first, address FROM emp
# DBi will write
#     Column ssn to Variable ssn,
#     Column last to Variable var_x,
#     Column first to Variable first, and
#     Column address to Variable var_y.

# Note how the mappings change when the columns are
# listed in another order.
sql SELECT last, first, address, ssn FROM emp
# DBi will write
#     Column last to Variable last,
#     Column first to Variable var_x,
#     Column address to Variable address, and
#     Column ssn to Variable var_y.
```

BINARY

define JAM/DB*i* variables for fetching binary values

SYNOPSIS

```
dbms BINARY variable [, variable ...]
```

DESCRIPTION

Many engines support a binary datatype for bytes strings and other non-printable data. An application cannot fetch binary values to JAM variables (fields, LDB variables, or JPL variables) but it may fetch them to JAM/DB*i* variables defined with the command DBMS BINARY.

variable is the name of the binary variable which JAM/DB*i* will create. Its definition may include a number of occurrences and/or a length. If a number of occurrences is supplied, it must be enclosed in square brackets. If a variable length is supplied, it must be enclosed in parentheses. If both are supplied, the number of occurrences must be first. Any of the following are permitted:

```
dbi_binvar
dbi_binvar [10] (255)
dbi_binvar [5]
dbi_binvar (8)
```

Any valid JAM variable name is a legal JAM/DB*i* variable name. The default number of occurrences is 1, and the default length is the maximum, 255. Memory is allocated for the occurrences when they are used (i.e., when a binary column is fetched).

If an application is selecting a binary column, use this command to create a binary variable for the column. The variable may have the same name as the column, or it may be mapped to the column with DBMS ALIAS. Because a binary variable is a target of a SELECT, JAM/DB*i* will examine its number of occurrences when determining how many rows to fetch. Therefore, the binary variable should have the same number of occurrences as the other JAM target variables. When searching for target variables, JAM/DB*i* searches among the binary variables before searching among the JAM variables. The developer is responsible for ensuring that the binary variable names do not conflict with JAM variable names.

The only legal use of a binary variable in JPL is in the USING clause of a DBMS EXECUTE statement. If no occurrence is given for the variable, the first occurrence is the default.

Once defined, a binary variable is available to the rest of the application. Note that

```
dbms BINARY dbi_binvar[10]
dbms BINARY timestamp[100]
```

is the same as

```
dbms BINARY dbi_binvar[10]  timestamp[100]
```

To delete all binary variables, execute DBMS BINARY with no arguments:

```
dbms BINARY
```

Several JAM/DB*i* library routines are provided for accessing and manipulating the binary variables. These routines are only available in C. They are described in Chapter 15. and listed below.

RELATED FUNCTIONS

```
dm_bin_create_occur (variable, occurrence);
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

EXAMPLE

```
# "photo" is a binary column
dbms BINARY dbi_binvar
dbms ALIAS photo dbi_binvar
sql SELECT jobcode, site, photo FROM newbuildings \
   WHERE architect = :+lastname


dbms BINARY lastchanged[20]
sql SELECT id, name, description,
```

CATQUERY

concatenate a full result row to a JAM variable or a file

SYNOPSIS

```
dbms [WITH CURSOR cursor] CATQUERY TO jamvar
   [SEPARATOR "text"] [HEADING [ON | OFF]]

dbms [WITH CURSOR cursor] CATQUERY TO FILE file
   [SEPARATOR "text"] [HEADING [ON | OFF]]
```

DESCRIPTION

The result columns of a SELECT statement are usually mapped to individual variables. Use CATQUERY to map full result rows to a variable's occurrences or to a text file. The options are described below.

WITH CURSOR ***cursor***	Names a declared SELECT cursor. If the clause is not used, JAM/DB*i* uses the default SELECT cursor.
TO ***jamvar***	Names a JAM variable as the destination.
TO FILE ***file***	Names a text file as the destination. If the file already exists, it is overwritten when the SELECT is executed.
SEPARATOR ***"text"***	Specifies that JAM/DB*i* should use ***text*** to separate column values in a result row. The default is two blank spaces.
HEADING ON	Specifies that JAM/DB*i* should put a heading at the beginning of the SELECT results. This is the default for a catquery to a file. The heading is built using the column names or any aliases assigned to the cursor. The maximum length of a heading is 255 characters. Any additional characters are truncated.
HEADING OFF	Specifies that JAM/DB*i* should not use a heading. This is the default for a catquery to a JAM variable.

JAM/DB*i* attempts to format the column values by searching for JAM variables of the same name and using their attributes for length, precision, and date-time or currency edits. The application may override any default formatting with the command DBMS FORMAT.

The catquery for a cursor is turned off by executing the DBMS CATQUERY command with no arguments. Closing a cursor also turns off the catquery. If a cursor is redeclared without being closed, the cursor keeps the catquery destination as the cursor's SELECT destination.

Catquery to a Variable

When the catquery destination is a JAM variable, JAM/DB*i* concatenates a result row and writes it to ***jamvar*** when the SELECT is executed. If ***jamvar*** is an LDB or field array, JAM/DB*i* writes the result rows to the array occurrences. If there are more result rows than occurrences in ***jamvar***, use DBMS CONINUE to fetch the additional rows.

If the clause HEADING ON is used, JAM/DB*i* creates a heading by using the cursor's aliases and column names. If ***jamvar*** has two or more occurrences, JAM/DB*i* will put the heading in the first occurrence of ***jamvar***.

Catquery to a Text File

When the catquery destination is a text file, JAM/DB*i* writes all the result rows to the specified text file when the SELECT is executed. Any existing file with the same name is overwritten. If a result row is longer than the page width, JAM/DB*i* wraps the row to the next line.

If aliases have been specified for the cursor, JAM/DB*i* uses those aliases as column headings in the text file. If there are no aliases, JAM/DB*i* uses the columns' names. If the clause HEADINGOFF is used, JAM/DB*i* does not print a heading.

Since all result rows are written to the file, the DBMS CONTINUE commands should not be used with a CATQUERY TO FILE cursor while the file is open.

The file remains open until DBMS CATQUERY is reset or the cursor is closed.

RELATED FUNCTIONS

dbms [WITH CURSOR ***cursor***] ALIAS [***column***] ***"text"*** ...

dbms [WITH CURSOR ***cursor***] FORMAT [***column***] ***format*** ...

EXAMPLE

```
# select an employee's first and last name
# and concatenate the values in the field "fullname"
 dbms DECLARE name_cursor CURSOR FOR \
   SELECT last, first WHERE ssn=:+ssn
 dbms WITH CURSOR name_cursor CATQUERY TO fullname \
   SEPARATOR ", "
 dbms WITH CURSOR name_cursor EXECUTE

# select the maximum value from the column "cost"
# and write it to the JPL variable "hi_cost"
# formatting it with currency edit saved with the LDB
# variable "money_var"
 vars hi_cost
 dbms DECLARE max_cursor CURSOR FOR \
```

```
    SELECT MAX(cost) FROM inventory
  dbms WITH CURSOR max_cursor CATQUERY TO hi_cost
  dbms WITH CURSOR max_cursor FORMAT money_var
  dbms WITH CURSOR max_cursor EXECUTE

# Write the results of the default SELECT cursor
# to a file with heading. Turn off ALIAS and CATQUERY
# when finished.
  dbms CATQUERY TO FILE phonelist
  dbms ALIAS emplast "Last Name", empfirst "First Name",\
    phone1 "Main Number", phone2 "Additional Number"
  sql SELECT emplast, empfirst, phone1, phone2 FROM emp
  dbms CATQUERY
  dbms ALIAS
```

CLOSE_ALL_CONNECTIONS

close all connections on an engine

SYNOPSIS

```
dbms CLOSE_ALL_CONNECTIONS
```

DESCRIPTION

When this command is executed, JAM/DB*i* closes every connection which the application declared on any and all engines. For each connection, it closes all cursors belonging to the connection, disconnects from the engine, and frees all structures associated with the connection.

SEE ALSO

JAM/DB*i* *Developer's Guide*, page 55.

VARIANTS

```
dbms CLOSE CONNECTION [connection]
```

RELATED FUNCTIONS

```
dbms [WITH ENGINE engine] DECLARE connection CONNECTION \
   FOR [OPTION arg ...]
```

CLOSE CONNECTION

close a declared connection

SYNOPSIS

```
dbms CLOSE CONNECTION [connection]
```

DESCRIPTION

Executing this command closes all open cursors associated with the named or default connection, logs off the connection from its engine, and frees the connection data structure.

SEE ALSO

JAM/DB*i* *Developer's Guide*, 55.

VARIANTS

```
dbms [WITH ENGINE engine] CLOSE_ALL_CONNECTIONS
```

RELATED FUNCTIONS

```
dbms [WITH ENGINE engine] DECLARE connection CONNECTION \
   FOR [OPTION arg ...]
```

```
WITH CONNECTION connection
```

CLOSE CURSOR

close a named or default cursor

SYNOPSIS

```
dbms CLOSE CURSOR [cursor]

dbms WITH CONNECTION connection CLOSE CURSOR
```

DESCRIPTION

Use this command to close an open cursor. Closing a cursor frees all structures associated with the cursor.

Closing a cursor is convenient way of turning off all attributes assigned to the cursor with DBMS ALIAS, DBMS CATQUERY, DBMS FORMAT, DBMS OCCUR, DBMS START, DBMS STORE_FILE, DBMS TYPE, and DBMS UNIQUE.

If ***cursor*** is not given, JAM/DB*i* closes the default SELECT cursor. A connection may also be specified when closing a default cursor. JAM/DB*i* will automatically declare another default SELECT cursor when needed. A connection name should not be given when closing a named cursor.

Closing a connection also closes all cursors associated with the connection.

SEE ALSO

JAM/DB*i* *Developer's Guide*, page 57.

VARIANTS

```
dbms [WITH ENGINE engine] CLOSE CONNECTION [connection]

dbms CLOSE_ALL_CONNECTIONS
```

RELATED FUNCTIONS

```
dbms [WITH CONNECTION connection] DECLARE cursor CURSOR \
   FOR SQLstmt

dbms WITH CURSOR cursor EXECUTE

WITH CURSOR cursor
```

EXAMPLE

```
# Assign a catquery and aliases to the default SELECT
 # cursor. Close the cursor when finished.
 dbms CATQUERY TO FILE phonelist
 dbms ALIAS emplast "Last Name", empfirst "First Name",\
   phone1 "Main Number", phone2 "Additional Number"
 sql SELECT emplast, empfirst, phone1, phone2 FROM emp
 dbms CLOSE CURSOR
```

CONNECTION

set or change the default connection

SYNOPSIS

dbms CONNECTION ***connection***

DESCRIPTION

If an application has declared two or more connections, the application may set a default connection with this command. The default connection is used for all subsequents statements that do not use a `WITH CONNECTION` or `WITH CURSOR` clause.

RELATED FUNCTIONS

dbms CLOSE CONNECTION ***connection***

dbms [WITH ENGINE ***engine***] DECLARE CONNECTION ***connection***

WITH CONNECTION ***connection***

EXAMPLE

```
dbms ENGINE sybase
dbms DECLARE a CONNECTION FOR \
   USER :uname PASSWORD :pword SERVER s1 DATABASE master
dbms DECLARE b CONNECTION FOR \
   USER :uname PASSWORD :pword SERVER s2 DATABASE projects
dbms CONNECTION a
dbms WITH CONNECTION b DECLARE c1 CURSOR FOR \
   INSERT finance (number, title, manager) \
   VALUES (::number, ::title, ::manager)
```

CONTINUE

fetch the next set of rows associated with a default or named SELECT cursor

SYNOPSIS

```
dbms [WITH CURSOR cursor] CONTINUE
```

DESCRIPTION

If a SELECT retrieves more rows than will fit in its destination variables, JAM/DB*i* will return only as many rows as will fit. It continues fetching more rows from the SELECT set when the application executes this command. If there are pending rows, executing this command clears the destination variables, and fetches the next screenful of rows from the SELECT set. If there are no pending rows, executing this command does nothing.

DBMS CONTINUE is always associated with a SELECT cursor. If no cursor is named, JAM/DB*i* uses the default SELECT cursor.

Note that if the cursor's aliases have changed between the execution of the SELECT and the execution of DBMS CONTINUE, DBMS CONTINUE uses the new settings.

This command should not be used with a CATQUERY TO FILE cursor. CATQUERY TO FILE always writes out the entire select set to the CATQUERY file.

VARIANTS

```
dbms [WITH CURSOR cursor] CONTINUE_DOWN
```

RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE_BOTTOM
dbms [WITH CURSOR cursor] CONTINUE_TOP
dbms [WITH CURSOR cursor] CONTINUE_UP
dbms [WITH CURSOR cursor] STORE [FILE [file]]
```

EXAMPLE

```
dbms DECLARE movie_list CURSOR FOR \
   SELECT * FROM movie_archive WHERE subject=::subj

proc get_selection
 dbms WITH CURSOR movie_list EXECUTE USING subject
```

```
jpl check_count
return


proc check_count
# DM_NO_MORE_ROWS is an LDB constant equal to 53256
if @dmretcode != DM_NO_MORE_ROWS
  msg setbkstat "Press %KPF1 to see more films " \
     "or press %KPF2 to specify another topic."
else
  msg setbkstat "That's all folks!"
return

proc get_more
# This function is called by pressing PF1.
# It retrieves the next set of rows.
dbms WITH CURSOR movie_list CONTINUE
jpl check_count
return
```

CONTINUE_BOTTOM

fetch the last page of rows associated with the default or named SELECT cursor

SYNOPSIS

```
dbms [WITH CURSOR cursor] CONTINUE_BOTTOM
```

DESCRIPTION

This command fetches the last screenful of rows from the cursor's SELECT set. If no cursor is named, JAM/DB*i* uses the default SELECT set. If number of rows in the SELECT set is less than the number of occurrences in the JAM variables, JAM/DB*i* will ignore the request.

Some engines automatically support this command. Other engines require a temporary storage file created by the command DBMS STORE FILE. If JAM/DB*i* returns DM_BAD_CMD error when the application executes this command, the engine needs a scrolling file. See the engine-specific *Notes* for more information.

This command should not be used with a CATQUERY TO FILE cursor.

RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE
dbms [WITH CURSOR cursor] CONTINUE_DOWN
dbms [WITH CURSOR cursor] CONTINUE_TOP
dbms [WITH CURSOR cursor] CONTINUE_UP
dbms [WITH CURSOR cursor] STORE [FILE [filename]]
```

EXAMPLE

```
#Engines not requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor EXECUTE
....
dbms WITH CURSOR emp_cursor CONTINUE_BOTTOM

#Engines requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor STORE FILE
dbms WITH CURSOR emp_cursor EXECUTE
....
dbms WITH CURSOR emp_cursor CONTINUE_BOTTOM
```

CONTINUE _DOWN

fetch the next set of rows associated with the default or named SELECT cursor

SYNOPSIS

```
dbms [WITH CURSOR cursor] CONTINUE_DOWN
```

DESCRIPTION

This command is identical to DBMS CONTINUE.

Note that CONTINUE is always associated with a SELECT cursor. If no cursor is named, JAM/DB*i* uses the default SELECT cursor.

VARIANTS

```
dbms [WITH CURSOR cursor] CONTINUE
```

RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE_BOTTOM
dbms [WITH CURSOR cursor] CONTINUE_TOP
dbms [WITH CURSOR cursor] CONTINUE_UP
dbms [WITH CURSOR cursor] STORE [FILE [filename]]
```

EXAMPLE

```
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor EXECUTE
....
proc get_more
dbms WITH CURSOR emp_cursor CONTINUE_DOWN
```

CONTINUE_TOP

fetch the first page of rows associated with the default or named SELECT cursor

SYNOPSIS

```
dbms [WITH CURSOR cursor] CONTINUE_TOP
```

DESCRIPTION

This command fetches the first screenful of rows from the cursor's SELECT set. If no cursor is named, JAM/DB*i* uses the default SELECT cursor. If number of rows in the SELECT set is less than the number of occurrences in the **JAM** variables, JAM/DB*i* will ignore the request.

Some engines automatically support this command. Other engines require a temporary storage file created by the command DBMS STORE FILE. If the engine needs such a file and the application tries to execute DBMS CONTINUE_TOP without executing DBMS STORE, JAM/DB*i* returns the error DM_BAD_CMD. See the engine-specific *Notes* for more information.

RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE

dbms [WITH CURSOR cursor] CONTINUE_BOTTOM

dbms [WITH CURSOR cursor] CONTINUE_DOWN

dbms [WITH CURSOR cursor] CONTINUE_UP

dbms [WITH CURSOR cursor] STORE [FILE [filename]]
```

EXAMPLE

```
#Engine not requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor EXECUTE
....
proc go_to_start
dbms WITH CURSOR emp_cursor CONTINUE_TOP
```

```
#Engines requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor STORE FILE
dbms WITH CURSOR emp_cursor EXECUTE
....
proc go_to_start
dbms WITH CURSOR emp_cursor CONTINUE_TOP
```

CONTINUE_UP

fetch the previous page of rows associated with the default or named SELECT cursor

SYNOPSIS

```
dbms [WITH CURSOR cursor] CONTINUE_UP
```

DESCRIPTION

Use this command to scroll backwards through a SELECT set. If no cursor is named, JAM/DB*i* uses the default SELECT set. If number of rows in the SELECT set is less than the number of occurrences in the **JAM** variables, JAM/DB*i* will ignore the request.

Some engines automatically support this command. Other engines require a temporary storage file created by the command DBMS STORE FILE. If the engine needs such a file and the application tries to execute DBMS CONTINUE_UP before executing DBMS STORE, JAM/DB*i* returns the error DM_BAD_CMD. See the engine-specific *Notes* for more information.

This command should not be used with a CATQUERY TO FILE cursor.

RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE

dbms [WITH CURSOR cursor] CONTINUE_BOTTOM

dbms [WITH CURSOR cursor] CONTINUE_DOWN

dbms [WITH CURSOR cursor] CONTINUE_TOP

dbms [WITH CURSOR cursor] STORE [FILE [filename]]
```

EXAMPLE

```
#Engine not requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor EXECUTE
....
proc go_back
dbms WITH CURSOR emp_cursor CONTINUE_UP
```

```
#Engines requiring STORE FILE
dbms DECLARE emp_cursor FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor STORE FILE
dbms WITH CURSOR emp_cursor EXECUTE
....
proc go_back
dbms WITH CURSOR emp_cursor CONTINUE_UP
```

DECLARE CONNECTION

create a named connection to a server and database

SYNOPSIS

```
dbms [WITH ENGINE engine] DECLARE connection CONNECTION \
    [FOR OPTION arg ...]
```

DESCRIPTION

Applications which must connect to two or more servers should use this command to declare a named connection to a server. If JAM/DB*i* executes this statement successfully, it allocates a connection structure and adds it to the list of open structures.

The combination of necessary or supported options is engine-specific. See the engine-specific *Notes* in this document for a listing.

The connection remains open until it is closed with `DBMS CLOSE CONNECTION` or `DBMS CLOSE_ALL_CONNECTIONS`.

RELATED FUNCTIONS

`dbms CLOSE CONNECTION [connection]`

`dbms CLOSE_ALL_CONNECTIONS`

`dbms CONNECTION connection`

`WITH CONNECTION connection`

DECLARE CURSOR

declare a named cursor for a SQL statement

SYNOPSIS

```
dbms [WITH CONNECTION connection] DECLARE cursor CURSOR \
   FOR SQLstmt
```

DESCRIPTION

Use this command to create or redeclare a named cursor.

If the application has not already declared ***cursor***, JAM/DB*i* allocates a new cursor structure and adds its name to the list of declared cursors.

If a structure already exists for ***cursor*** and the connection is the same, JAM/DB*i* reinitializes the structure. Reinitialization clears any information on `SELECT` columns, binding parameters, and the maximum number of rows to fetch. It does not clear any attributes assigned to the cursor with the statements `DBMS ALIAS`, `DBMS CATQUERY`, `DBMS FORMAT`, `DBMS OCCUR`, `DBMS START`, `DBMS STORE`, `DBMS TYPE`, or `DBMS UNIQUE`. JAM/DB*i* will use these settings if the cursor is redeclared with a `SELECT` statement. It will ignore the attributes if the cursor is redeclared with an `INSERT`, `UPDATE`, or `DELETE` statement. To redeclare the cursor with a new (empty) structure, close the cursor with `DBMS CLOSE CURSOR` before executing the new declaration.

If a cursor is redeclared on another connection, JAM/DB*i* automatically closes the cursor and declares a new structure.

The cursor remains open until it is explicitly closed with the command `DBMS CLOSE CURSOR`. Closing a connection also closes all cursors on the connection.

There are few restrictions on valid cursor names. However, you should avoid using any SQL or JAM/DB*i* keyword as a cursor name. Please note that JAM/DB*i* is case sensitive regarding cursor names; for example, it interprets cursor `c1` as distinct from cursor `C1`.

SEE ALSO

JAM/DB*i* *Developer's Guide*, pages 57, 72.

RELATED FUNCTIONS

`dbms WITH CURSOR cursor EXECUTE`

`dbms CLOSE CURSOR cursor`

`WITH CURSOR cursor`

EXAMPLE

```
dbms WITH ENGINE oracle DECLARE emp_cursor FOR \
   SELECT ss, last, first FROM emp \
   WHERE dept = ::parameter
...
dbms WITH CURSOR emp_cursor EXECUTE USING dept_name
```

ENGINE

set or change the default engine

SYNOPSIS

dbms ENGINE ***engine***

DESCRIPTION

If an application has initialized two or more engines, the application may use this command to set a default engine. If an application has only one initialized engine, JAM/DB*i* automatically assigns that engine as the default.

engine is a mnemonic associated with one of the support routines initialized in `dbiinit.c` or in a call to `dm_init`.

SEE ALSO

JAM/DB*i* *Developer's Guide*, page 52.

RELATED FUNCTIONS

WITH ENGINE ***engine***

EXECUTE

execute the SQL statement declared for a named cursor

SYNOPSIS

```
dbms WITH CURSOR cursor EXECUTE [USING args]
```

DESCRIPTION

Use this statement to execute the statement associated with a declared cursor.

This statement does not support the WITH CONNECTION clause. JAM/DB*i* uses the engine that was specified either by name or by default when the cursor was declared. The only way to change the cursor's engine or connection is to redeclare the cursor.

If an application is executing a similar statement many times, it is often more efficient to declare a cursor for the statement. Usually the engine saves the parsed statement, executing it when the application executes the cursor. It is not necessary to redeclare the cursor to supply new data for a WHERE or VALUES clause. Instead, the application may declare the cursor and use a substitution parameter for each value that the application will supply when it executes the cursor. Substitution parameters begin with a double colon (::). For example,

```
dbms DECLARE c1 CURSOR FOR \
   SELECT * FROM titles WHERE author LIKE ::author_parm
```

author_parm is simply a place holder for the value that will be supplied when the cursor is executed. For example,

```
dbms WITH CURSOR c1 EXECUTE USING "Fau%"
```

would fetch rows where author began with the characters "Fau." The application could execute the cursor repeatedly, each time with a new value. It may use the value of a field to supply a value. For example,

```
dbms WITH CURSOR c1 EXECUTE USING aname
```

Since aname is not quoted, JAM/DB*i* assumes it is a JAM variable. If an argument in the USING clause is a field or LDB variable with a date-time, currency, null field, or type edit JAM/DB*i* formats the variable's value before passing it to the engine.

This topic is covered in detail in the *Developer's Guide*.

SEE ALSO

JAM/DB*i* *Developer's Guide*, page 72.

RELATED FUNCTIONS

```
dbms DECLARE cursor CURSOR FOR SQLstmt
```

```
dbms CLOSE CURSOR cursor

dbms [WITH CURSOR cursor] CONTINUE

WITH CURSOR cursor
```

EXAMPLE

```
  dbms DECLARE x CURSOR FOR \
    SELECT * FROM inventory WHERE lname=::p1 OR ss=::p2

# bind by position:
 dbms WITH CURSOR x EXECUTE USING newname, ss_number

 # or bind by name:
 dbms WITH CURSOR x EXECUTE \
   USING p1 = newname, p2 = ss_number
```

FORMAT

format CATQUERY values

SYNOPSIS

```
dbms [WITH CURSOR cursor] FORMAT \
   [[column] format [, [column] format ...]]
```

DESCRIPTION

Use this command to format CATQUERY values before writing them to a variable or a text file. The options are explained below.

WITH CURSOR ***cursor***	Names a declared SELECT cursor. If the clause is not used, JAM/DB*i* uses the default SELECT cursor.
column	Names a selected column. The case of ***column*** should match the setting of the case flag for the engine in `dbiinit.c`. If columns are not named, the formats are applied by position.
format	Describes how JAM/DB*i* should format the value. ***format*** is either a JAM variable or a quoted precision edit.

If ***format*** is a JAM variable, JAM/DB*i* formats the column value as if it were writing to the field. In particular, the following characteristics will affect the formatting:

- variable's maximum shifting length
- variable's JAM type

See Section 9.1.3. in the *Developer's Guide* of this document for more information about formatting with JAM type.

format may also be a precision edit. A precision edit is a quoted string beginning with a percent sign. It supplies the length of the value, and optionally, a decimal precision for numeric values.

A precision is given in the form

"%width"

"%width.precision"

To turn off formatting on the default or named cursor, execute the command with no arguments.

EXAMPLE

```
# use column "lastname" exactly as returned
# format column "revdate" with the LDB variable "today",
# format column "sal" to width 15 with 2 decimal places,
# format column "comment" to width 30 and truncate excess
dbms CATQUERY TO FILE listing
dbms FORMAT revdate today, sal "%15.2", comment  "%30"
sql SELECT lastname, sal, revdate, comment FROM employee
```

OCCUR

change the behavior of a SELECT cursor that writes to JAM arrays

SYNOPSIS

```
dbms [WITH CURSOR cursor] OCCUR  occ_int [MAX int]
dbms [WITH CURSOR cursor] OCCUR CURRENT [MAX int]
```

DESCRIPTION

By default, if the destination of a SELECT is one or more arrays, JAM/DB*i* fetches as many rows as will fit in the arrays and begins writing at the first occurrence in the arrays. Use this command to change the default behavior for a SELECT cursor. The options for the command are:

WITH CURSOR ***cursor***	Names a declared SELECT cursor. If the clause is not used, JAM/DB*i* uses the default SELECT cursor.
occ_int	Specifies the occurrence number where JAM/DB*i* should begin placing SELECT results.
CURRENT	Specifies that JAM/DB*i* should use the occurrence number of the "current" field. JAM/DB*i* begins writing at this occurrence number in the target arrays. Note that the current field is the one containing the JAM screen cursor and is not necessarily a target variable.
MAX ***int***	Specifies the maximum number of rows to fetch for a SELECT or CONTINUE. If ***int*** is less than 1, no rows are fetched.

The setting is turned off by executing the DBMS OCCUR command with no arguments. Closing a cursor also turns off the setting. If a cursor is redeclared without being closed, the cursor continues to use to the setting for SELECT's and CONTINUE's.

DBMS OCCUR is ignored with a CATQUERY cursor.

RELATED FUNCTIONS

[WITH CURSOR ***cursor***]

EXAMPLE

```
dbms DECLARE title_cursor CURSOR FOR \
   SELECT * FROM booklist WHERE isbn = :+code
 dbms WITH CURSOR title_cursor OCCUR CURRENT
 dbms WITH CURSOR title_cursor EXECUTE
```

ONENTRY

install an entry function

SYNOPSIS

```
dbms ONERROR CALL function
dbms ONERROR JPL jpl_entry_point
```

DESCRIPTION

Use this command to install a JPL routine or a C function which JAM/DB*i* will call before it executes a `sql` or `dbms` statement.

Currently, this function is for informational purposes only. For instance, you may wish to log statements to a file on disk before executing them. You may use this function with an exit handler to track the start and complete time for a query or any database other operation.

The function is passed three arguments:

1. a copy of the first 255 characters of the statement; if the statement was executed from JPL, this is the first 255 characters after the command word `sql` or `dbms`
2. the name of the engine where
3. context flag; for the entry handler its value is 1.

The function's return code is not used.

If the error occurred while executing a JPL statement with the command `dbms` or `sql`:

- 0 returns control to the JPL procedure where the error occurred
- 1 aborts the JPL procedure where the error occurred and returns 1 to the procedure's caller (either JAM or another JPL procedure)

If the error occurred while executing a statement with one of the `dm_` library functions, the `dm_` function returns the error handler's return code.

To use a C function as an error handler, you must first install the function as a prototyped function. Please consult the JAM *Programmer's Guide* for more information.

SEE ALSO

JAM/DB*i* *Developer's Guide*, page 93.

JAM/DB*i* *Reference Guide*, global variables, page 113

RELATED FUNCTIONS

dbms ONEXIT [JPL *entrypoint* | CALL *function*]

ONERROR

set the behavior of the error handler

SYNOPSIS

```
dbms ONERROR CALL function
dbms ONERROR CONTINUE
dbms ONERROR JPL jpl_entry_point
dbms ONERROR STOP
```

DESCRIPTION

Use this command to set or change the behavior of the JAM/DB*i* error handler for the application. The default error handler displays an error message. The source of the message is determined by the engine's initialization. If an engine is initialized with the flag DM_DEFAULT_ENG_MSG the default error handler displays an engine-specific error message. If it is initialized with the flag DM_DEFAULT_DBI_MSG the error handler uses messages only from the JAM message file. If an error occurs while executing a JPL procedure, the default handler aborts the procedure, returning –1 to the calling procedure.

An application may override the default error handler with the command DBMS ONERROR and an argument. Please note that the error handler is global to the application. Each execution of this command overrides the previous error handler.

The command variants are explained below.

ONERROR STOP

This command restores the default error handler.

ONERROR CONTINUE

This command prevents the default error handler from aborting a JPL procedure where a JAM/DB*i* error occurs. Message display is not changed.

ONERROR JPL or ONERROR CALL

These commands install a user function as the error handler. If JAM/DB*i* or the engine find an error, JAM/DB*i* updates the global error and status variables (i.e., @dm variables) and calls the installed function.

The function displays any error messages and its return code controls whether or not JPL execution is aborted.

The function is passed three arguments:

1. the first 255 characters of the statement; if the statement was executed from JPL, this is the first 255 characters after the command word sql or dbms

2. the name of the engine for the attempted statement
3. context flag; for the error handler its value is 2.

The function's return code is returned to the application.

If the error occurred while executing a JPL statement with the command `dbms` or `sql`:

- 0 returns control to the JPL procedure where the error occurred
- 1 aborts the JPL procedure where the error occurred and returns 1 to the procedure's caller (either JAM or another JPL procedure)

If the error occurred while executing a statement with one of the `dm_` library functions, the `dm_` function returns the error handler's return code.

To use a C function as an error handler, you must first install the function as a prototyped function. Please consult the JAM *Programmer's Guide* for more information.

SEE ALSO

JAM/DB*i* *Developer's Guide*, page 93.

JAM/DB*i* *Reference Guide*, global variables, page 113

RELATED FUNCTIONS

`dbms ONEXIT [JPL` ***entrypoint*** `| CALL` ***function***`]`

ONEXIT

install an exit handler

SYNOPSIS

```
dbms ONEXIT CALL function
dbms ONERROR JPL jpl_entry_point
```

DESCRIPTION

Use this command to install a function which JAM/DB*i* will call after executing a `dbms` or `sql` command from JPL or C. You may use this function to process error and status codes after every command.

Installing an `ONEXIT` function will override the default error handler. Please note that the exit handler is global to the application. Each execution of this command overrides the previous exit handler.

The function is passed three arguments:

1. the first 255 characters of the statement; if the statement was executed from JPL, this is the first 255 characters after the command word `sql` or `dbms`
2. the name of the engine for the attempted statement
3. context flag; for the exit handler its value is 1.

The function's return code is returned to the application. If an error occurred while executing a JPL statement with the command `dbms` or `sql` and there is no `ONEXIT` function, then

- 0 returns control to the JPL procedure where the error occurred
- 1 aborts the JPL procedure where the error occurred and returns 1 to the procedure's caller (either JAM or another JPL procedure)

If an error occurred while executing a statement with one of the `dm_` library functions and there is no `ONEXIT` function, the `dm_` function returns the exit handler's return code.

To use a C function as an exit handler, you must first install the function as a prototyped function. Please consult the JAM *Programmer's Guide* for more information.

SEE ALSO

JAM/DB*i* *Developer's Guide*, page 93.

JAM/DB*i* *Reference Guide*, global variables, page 113

RELATED FUNCTIONS

dbms ONEXIT [JPL ***entrypoint*** | CALL ***function***]

START

specify a starting row in a SELECT set

SYNOPSIS

```
dbms [WITH CURSOR cursor] START [int]
```

DESCRIPTION

By default, when a SELECT set contains more than one row, JAM/DB*i* fetches them sequentially beginning with the first row in the SELECT set. Use this command to begin fetching at row ***int***. JAM/DB*i* will read and discard ***int*** – 1 rows from the SELECT set before returning the requested rows to the application. If the application is counting the rows fetched, the discarded rows do not update `@dmrowcount`. If ***int*** is greater than the number of rows in the SELECT set, no rows are displayed.

If no cursor is specified, JAM/DB*i* uses the default SELECT cursor.

The setting is turned off by executing DBMS START with no arguments. Closing a cursor also turns off the setting. If a cursor is redeclared without being closed, the cursor continues to use to the setting for SELECT's.

RELATED FUNCTIONS

`WITH CURSOR` ***cursor***

EXAMPLE

```
proc discard_100
# dbi_count is an LDB variable
dbms COUNT dbi_count
dbms START 100
sql SELECT * FROM emp
if @dmrowcount == 0
   msg emsg "There are less than 100 employees."
dbms START
return
```

STORE

set up a continuation file for a named or default cursor

SYNOPSIS

```
dbms [WITH CURSOR cursor] STORE [FILE [filename]]
```

DESCRIPTION

When this command is used with a SELECT cursor, JAM/DB*i* maintains a copy of the result rows in a temporary binary file. The use of a file permits an application to scroll forward and backward in a SELECT set, even if the database has no native support for backward scrolling.

If ***filename*** is not given, JAM/DB*i* calls the standard C library routine `tmpfile` to create and open a temporary binary file.

A continuation file remains open for the life of the cursor, or until the feature is turned off with the command,

```
dbms [WITH CURSOR cursor] STORE
```

Executing the command without the keyword FILE closes and deletes the file and turns off the feature for the named or default cursor. Closing the cursor also closes and deletes the file. If a cursor is not closed but simply redeclared for another SELECT statement, the file is cleared. Therefore, a continuation file holds the results of one SELECT statement only.

The use of a continuation file does not force the engine to return the entire SELECT set when the SELECT is executed. In its usual manner, JAM/DB*i* examines the number of occurrences in the destination variable to determine the number of rows to fetch. Each time it fetches rows from the engine by executing the SELECT or a DBMS CONTINUE, JAM/DB*i* updates the screen and appends the new data to the continuation file. If the application wishes to see rows already fetched, JAM/DB*i* uses the continuation file to get the rows and update the screen. If JAM/DB*i* reaches the end of the continuation file and the application executes another DBMS CONTINUE, JAM/DB*i* will attempt to get more rows from the engine. When the engine returns the no-more-rows code, JAM/DB*i* sets `@dmretcode` to the value of `DM_NO_MORE_ROWS`. Similarly, if the application attempts to scroll back past the first row in the continuation file, JAM/DB*i* sets `@dmretcode` to `DM_NO_MORE_ROWS`. See Appendix B. for a list of error and status codes. Write errors are not reported.

This command provides several advantages:

- a means for displaying very large SELECT sets without keeping all rows in memory at once

- better response time for very large SELECT sets; since fetches are incremental it is not necessary to get the entire SELECT set at once
- a means for forcing an engine to release a shared lock on a large SELECT set

Consult the *Notes* for information on engine-specific scrolling issues.

RELATED FUNCTIONS

```
dbms [WITH CURSOR cursor] CONTINUE_BOTTOM

dbms [WITH CURSOR cursor] CONTINUE_TOP

dbms [WITH CURSOR cursor] CONTINUE_UP
```

EXAMPLE

```
dbms DECLARE emp_cursor CURSOR FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor STORE FILE
dbms WITH CURSOR emp_cursor EXECUTE
jpl mapkeys

proc mapkeys
vars SPGU(6) SPGD(6) APP1(6) APP2(6) XLATE(1)
cat SPGU "0x113"
cat SPGD "0x114"
cat APP1 "0x6102"
cat APP2 "0x6202"
cat XLATE "2"
# Set the control strings for APP1 and APP2 on
# this screen to call DBi scroll functions
call sm_putjctrl :APP1 "^jpl scroll_forward" 0
call sm_putjctrl :APP2 "^jpl scroll_back" 0
# Remap the logical page up and down keys to
# APP1 and APP2. (This should be reset on screen exit.)
call sm_keyoption :SPGU :XLATE :APP1
call sm_keyoption :SPGD :XLATE :APP2
return

proc scroll_forward
# SPGU -> APP1 = ^jpl scroll_forward
dbms WITH CURSOR emp_cursor CONTINUE
return
```

```
proc scroll_back
# SPGD -> APP2 = ^jpl scroll_back
dbms WITH CURSOR emp_cursor CONTINUE_UP
return
```

UNIQUE

suppress repeating values in selected columns

SYNOPSIS

```
dbms [WITH CURSOR cursor] UNIQUE column [, column...]
```

DESCRIPTION

The following command suppresses repeating values in each named column of a SELECT set when the values are in adjacent rows. Typically, this feature is set for a column named in an ORDER BY clause.

The options are

WITH CURSOR ***cursor***	Names a declared SELECT cursor. If the clause is not used, JAM/DB*i* uses the default SELECT cursor.
column	Specifies a column name in the SELECT statement.

If no cursor is specified, JAM/DB*i* uses the default SELECT cursor.

If the destination variable has a null edit, an occurrence containing a suppressed value is blank, not null.

The setting is turned off by executing the DBMS UNIQUE command with no arguments. Closing a cursor also turns off the setting. If a cursor is redeclared without being closed, the cursor continues to use to the setting for SELECT's and CONTINUE's.

RELATED FUNCTIONS

WITH CURSOR ***cursor***

EXAMPLE

```
#Since several items may be ordered on the same invoice,
#suppress repeating invoice numbers when listing
#outstanding sales orders.

dbms DECLARE order_cursor CURSOR FOR \
   SELECT invoice_no, id, desc, quan, cost FROM newsales \
   ORDER BY invoice_no
dbms WITH CURSOR order_cursor UNIQUE invoice_no
dbms WITH CURSOR order_cursor EXECUTE
```

WITH CONNECTION

use a named connection for the duration of a statement

SYNOPSIS

```
dbms WITH CONNECTION connection DBMS_statement...
sql WITH CONNECTION connection SQL_statement ...
```

DESCRIPTION

This clause specifies a connection for the execution of the command, overriding the default connection. ***connection*** must be declared and open.

Any `sql` statement may use this clause.

Some `dbms` statements may also use it. In particular,

```
dbms [WITH CONNECTION connection] DECLARE cursor CURSOR...
```

Once a cursor is declared it remains associated with the connection on which it was declared. After declaring the cursor, the `WITH CONNECTION` clause should not be used in statements that manipulate the cursor. However, the `WITH CONNECTION` clause may be used on statements that manipulate the default cursor on any declared connection. Therefore, the following statements:

```
dbms WITH CONNECTION connection ALIAS ...
dbms WITH CONNECTION connection CATQUERY ...
dbms WITH CONNECTION connection CLOSE CURSOR
dbms WITH CONNECTION connection CONTINUE
dbms WITH CONNECTION connection CONTINUE_BOTTOM
dbms WITH CONNECTION connection CONTINUE_TOP
dbms WITH CONNECTION connection CONTINUE_UP
dbms WITH CONNECTION connection FORMAT ...
dbms WITH CONNECTION connection OCCUR ...
dbms WITH CONNECTION connection START ...
dbms WITH CONNECTION connection STORE ...
dbms WITH CONNECTION connection UNIQUE ...
```

perform the request on the default `SELECT` cursor on the named connection.

Some engine-specific `dbms` commands may also support the `WITH CONNECTION` clause. See the engine-specific *Notes* for more information.

SEE ALSO

JAM/DB*i* *Developer's Guide*, page 55.

Engine-specific *Notes*.

RELATED FUNCTIONS

dbms [WITH ENGINE ***engine***] DECLARE ***connection*** CONNECTION \
SERVER ***server*** [DB ***database***]

dbms CONNECTION ***connection***

dbms CLOSE CONNECTION [***connection***]

dbms CLOSE_ALL_CONNECTIONS

WITH CURSOR ***cursor***

WITH ENGINE ***engine***

WITH CURSOR

use a named cursor for the duration of a statement

SYNOPSIS

dbms WITH CURSOR ***cursor DBMS_statement***

DESCRIPTION

This clause specifies the name of a declared cursor on which JAM/DB*i* will execute the dbms command.

Once a cursor has been declared, the application may manipulate or execute the cursor by using the WITH CURSOR clause.

```
dbms WITH CURSOR cursor ALIAS ...
dbms WITH CURSOR cursor CATQUERY ...
dbms WITH CURSOR cursor CONTINUE
dbms WITH CURSOR cursor CONTINUE_BOTTOM
dbms WITH CURSOR cursor CONTINUE_TOP
dbms WITH CURSOR cursor CONTINUE_UP
dbms WITH CURSOR cursor EXECUTE ...
dbms WITH CURSOR cursor FORMAT ...
dbms WITH CURSOR cursor OCCUR ...
dbms WITH CURSOR cursor START ...
dbms WITH CURSOR cursor STORE ...
dbms WITH CURSOR cursor UNIQUE ...
```

If the WITH CURSOR clause is not used with these statements, JAM/DB*i* uses the default SELECT cursor. The application may also manipulate the default cursor by using the WITH CONNECTION clause.

Some engine-specific dbms commands may also support the WITH CONNECTION clause. See the engine-specific *Notes* for more information.

SEE ALSO

JAM/DB*i* *Developer's Guide*, page 57.

Engine-specific *Notes*.

RELATED FUNCTIONS

dbms DECLARE ***cursor*** CURSOR FOR ***SQLstmt***

```
dbms CLOSE CURSOR cursor
WITH CONNNECTION connection
WITH ENGINE engine
```

WITH ENGINE

use a named engine for the duration of a statement

SYNOPSIS

dbms WITH ENGINE ***engine DBi_command...***

DESCRIPTION

This clause specifies which engine JAM/DB*i* should use when executing a command. ***engine*** must be an initialized engine. An engine is initialized by using the `vendor_list` structure in `dbiinit.c` or by a call to `dm_init`.

engine must be one of the mnemonics associated with an initialized support routine.

The following commands accept an optional `WITH ENGINE` clause:

dbms WITH ENGINE ***engine*** DECLARE ***connection*** CONNECTION ...

If the `WITH ENGINE` clause is not used, JAM/DB*i* uses the default engine. If only one engine is initialized, that engine is automatically the default. An application using two or more engines may set the default engine with the `DBMS ENGINE` command.

Once a connection is declared it remains associated with the engine on which it was declared. After declaring the connection, the `WITH ENGINE` clause is no longer necessary or valid in any statement except `DBMS CLOSE CONNECTION` if the application wishes to close the default connection on an engine.

SEE ALSO

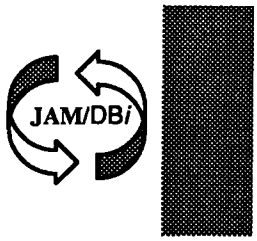
JAM/DB*i* *Developer's Guide*, 52.

RELATED FUNCTIONS

dbms ENGINE ***engine***

WITH CONNECTION ***connection***

WITH CURSOR ***cursor***



Chapter 15.
JAM/DB*i* *Library Reference*

This chapter contains a reference page for each of the JAM library functions.

The library includes functions for initializing JAM/DB*i*, and installing and de-installing an engine at runtime. The functions are:

- `dm_dbi_init`: initialize JAM for use with JAM/DB*i*.
- `dm_init`: initialize an engine.
- `dm_reset`: close all structures associated with an engine.

It includes functions for executing SQL and DBMS commands. The functions are

- `dm_dbms`: execute any DBMS command directly from C.
- `dm_sql`: execute any SQL statement directly from C.
- `dm_dbms_noexp`: like `dm_dbms` except no colon preprocessing is performed.
- `dm_sql_noexp`: like `dm_sql` except no colon preprocessing is performed.

It provides a function for simulating colon-plus processing from C. It is

- `dm_expand`

It provides a function for getting the full text of the last executed `dbms` or `sql` command. It is

- `dm_getdbitext`

The library also provides functions for handling binary values. They are

- `dm_bin_create_occur`
- `dm_bin_delete_occur`

- `dm_bin_get_dlength`
- `dm_bin_get_occur`
- `dm_bin_length`
- `dm_bin_max_occur`
- `dm_bin_set_dlength`

Developers may use these functions in any C hook function. Each reference page has the following sections:

- A synopsis of the function, including a listing of available keywords and arguments.
- A description of the function.
- A list of related functions.
- An example.

dm_bin_create_occur

get or allocate an occurrence in a binary variable

SYNOPSIS

```
char *dm_bin_create_occur (variable, occurrence)
char  *variable;
int    occurrence;
```

DESCRIPTION

If the application has created a binary variable with DBMS BINARY, this routine gets the specified occurrence from the variable. If the occurrence has not been allocated, this routine will allocate it. Note that ***occurrence*** must be less than or equal to the number of occurrences specified in the DBMS BINARY statement.

RETURNS

0 if the variable is not found or the occurrence number is not valid
else a pointer to an occurrence in a binary variable

VARIANTS

```
dm_bin_get_occur (variable, occurrence);
```

RELATED FUNCTIONS

```
dbms BINARY variable[occ] (length) [, variable [occ] (length)...]]
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

dm_bin_delete_occur

delete an occurrence in a binary variable

SYNOPSIS

```
void dm_bin_delete_occur (variable, occurrence)
char  *variable;
int   occurrence;
```

DESCRIPTION

If the application has created a binary variable with DBMS BINARY and the occurrence has been allocated, this routine frees the specified occurrence and sets the pointer to the occurrence to 0. If the occurrence has not been allocated, the routine does nothing.

RETURNS

Nothing.

RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]

dm_bin_create_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

dm_bin_get_dlength

get the length of an occurrence in a binary variable

SYNOPSIS

```
unsigned int dm_bin_get_dlength (variable, occurrence)
char  *variable;
int   occurrence;
```

DESCRIPTION

If the application has created a binary variable with DBMS BINARY and the occurrence has been allocated, this routine returns the length of the contents in the specified occurrence.

RETURNS

0 if variable or occurrence is not found,

else the length of the occurrence

RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]]

dm_bin_create_occur (variable, occurrence);
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

dm_bin_get_occur

get the data in an occurrence of a binary variable

SYNOPSIS

```
char *dm_bin_get_occur (variable, occurrence)
char  *variable;
int    occurrence;
```

DESCRIPTION

If the application has created a binary variable with DBMS BINARY and the occurrence has been allocated, this routine gets the specified occurrence from the variable.

RETURNS

0 if the variable or occurrence is not found
else a pointer to an occurrence in the variable

VARIANTS

```
dm_bin_create_occur (variable, occurrence);
```

RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

dm_bin_length

get the maximum length of an occurrence in a binary variable

SYNOPSIS

```
unsigned int dm_bin_length (variable)
char  *variable;
```

DESCRIPTION

If the application has created a binary variable with DBMS BINARY, this routine gets the maximum length of a single occurrence in the variable. To get the length of an occurrence's contents, use dm_bin_get_dlength.

RETURNS

0 if the variable is not found
else the length of the variable

RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]

dm_bin_create_occur (variable, occurrence);
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_max_occur (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

dm_bin_max_occur

get the maximum number of occurrences in a binary variable

SYNOPSIS

```
int dm_bin_max_occur (variable)
char  *variable;
```

DESCRIPTION

If the application has created a binary variable with DBMS BINARY, this routine gets the maximum number of occurrences in the variable.

RETURNS

0 if variable is not found
else the number of occurrences in the variable.

RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]

dm_bin_create_occur (variable, occurrence);
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_length (variable);
dm_bin_set_dlength (variable, occurrence, length);
```

dm_bin_set_dlength

set the length of an occurrence in a binary variable

SYNOPSIS

```
void dm_bin_set_dlength (variable, occurrence, length)
char   *variable;
int    occurrence;
unsigned int    length;
```

DESCRIPTION

If the application has created a binary variable with DBMS BINARY, this routine sets the maximum length of a single occurrence in the binary variable. ***length*** may be less than or greater than the variable's declared length.

RETURNS

Nothing.

RELATED FUNCTIONS

```
dbms BINARY [variable [, variable ...]

dm_bin_create_occur (variable, occurrence);
dm_bin_delete_occur (variable, occurrence);
dm_bin_get_dlength (variable, occurrence);
dm_bin_get_occur (variable, occurrence);
dm_bin_length (variable);
dm_bin_max_occur (variable);
```

dm_dbi_init

initialize JAM for JAM/DB*i*

SYNOPSIS

```
void dm_dbi_init ()
```

DESCRIPTION

JAM must be initialized for use with JAM/DB*i*. This function tells JAM the class of error messages for JAM/DB*i* and how to handle the JAM/DB*i* JPL commands `dbms` and `sql`.

In the distributed source files `jmain.c` and `jxmain.c`, this function is called in the `initialize` routine. Developers modifying these source files or using a custom executive, may call this routine at another time. `dm_dbi_init` should be called before `sm_initcrt` to ensure that the message file is loaded properly.

RETURNS

Nothing

dm_dbms

execute a DBMS command directly from C

SYNOPSIS

```
int dm_dbms (arg)
char *arg;
```

DESCRIPTION

Use this function to execute any DBMS command directly from C.

First *arg* is examined for the WITH ENGINE or WITH CONNECTION clause. If it is not used, dm_dbms assumes the default engine and connection. Next the colon preprocessor examines *arg* for colon variables. Finally, *arg* is passed to the appropriate routine for handing DBMS commands.

After executing the requested command, JAM/DB*i* updates all global status and error variables (@dm).

If the application has installed an entry function with DBMS ONENTRY, an exit function with DBMS ONEXIT, or an error handler with DBMS ONEXIT.the installed function will be called for commands executed through the function dm_dbms.

RETURNS

0 is no error
else an error code from the default or installed error handler

RELATED FUNCTIONS

dm_dbms_noexp (*arg*);

dm_sql (*arg*);

EXAMPLE

```
int start_up ()
{
    int retcode;

    retcode = dm_dbms ("ONERROR CALL do_error");
    if (retcode)
    {
        sm_emsg ("Cannot install the application error handler.")
        return 0;
```

```
    }
    dm_dbms ("DECLARE c1 CONNECTION FOR USER :user PASSWORD :password");

    return 0;
}
```

dm_dbms_noexp

execute a DBMS command without colon preprocessing

SYNOPSIS

```
int dm_dbms_noexp (arg)
char *arg;
```

DESCRIPTION

This function is identical to dm_dbms except that colon preprocessing is NOT performed on ***arg***.

RETURNS

0 is no error
else a return code from an installed or default error handler

RELATED FUNCTIONS

dm_dbms (***arg***);

dm_expand (***arg***);

dm_sql (***arg***);

dm_sql_noexp (***arg***);

dm_expand

format a string for an engine

SYNOPSIS

```
int dm_expand (engine, data, type, buf, buflen, edit)
char *arg;
char *data;
int type;
char *buf;
int *buflen;
char *edit;
```

DESCRIPTION

Use this function to format a string for a particular engine and JAM type. The function copies the formatted string to a buffer provided by the program.

engine is the name of an initialized engine. If this argument is null, JAM/DB*i* uses the default engine.

data is the string to format. Use a JAM library functions such as `sm_getfield` to get the value of a field or LDB entry.

type is one of the JAM types defined in `smedits.h`:

- `DT_CURRENCY`
- `DT_DATETIME`
- `DT_YESNO`
- `FT_CHAR`
- `FT_DOUBLE`
- `FT_INT`
- `FT_LONG`
- `FT_FLOAT`
- `FT_SHORT`

buf is a buffer provided by the program. The program is responsible for allocating a buffer large enough for the formatted string. ***buflen*** points to the size of the buffer. Upon return

from dm_expand, the value contained in the integer will be the length of the formatted text. The program can compare this value with the allocated length to ensure that truncation did not occur.

edit is a date-time edit string describing ***data***. It is required when type is DT_DATETIME. Use sm_edit_ptr to get a format from a date-time field, or construct a format string using JAM 's date-time tokens. See sm_dtime for more information.

RETURNS

0 is no error,
–1 if ***engine*** is invalid,
–2 if arguments are invalid (illegal JAM type, ***buflen*** <= 0, ***buf*** not allocated, or DT_DATETIME was used without a datetime edit)

–3 formatting routine failed

RELATED FUNCTIONS

```
int dm_dbms_noexp (arg);
```

```
int dm_sql_noexp (arg);
```

EXAMPLE

```
#include "smdefs.h"
#include "smedits.h"
#include "smerror.h"

#define   FLD_NOT_FOUND -1;
#define   MALLOC_ERROR  -2;
#define   EXPAND_ERROR  -3;
#define   NO_FORMAT     -4;


}int
formatter (src_name, dst_name, engine, jamtype)
char *src_name, *dst_name, *engine;
int jamtype;
{
   int dst_len, src_len, prec, ret;
   char *edit, *dst_buf, *src_buf;

            /* Get data. */
   /* Allocate a buffer based on the length of the source   */
   /* text and call getfield.                               */
   if ( (src_len = sm_n_dlength (src_name)) == -1)
      return FLD_NOT_FOUND;
```

```
    if ((src_buf=malloc(src_len + 1)) == 0)
        return MALLOC_ERROR;
    sm_n_getfield (src_buf, src_name);

    /* If no type was supplied, get it from the source field.*/
    if (jamtype == 0)
    {
        jamtype = sm_n_ftype(src_name, &prec);
    }

    /* If type is DT_DATETIME get format from field.   */
    if (jamtype == DT_DATETIME)
    {
        edit = sm_n_edit_ptr (src_name, UDATETIME);
        if (edit == 0)
        {
            edit = sm_n_edit_ptr (src_name, SDATETIME);
            if (edit == 0)
                return NO_FORMAT;
        }
        edit = edit +2;
    }

    /* Allocate a buffer based on the length of the
        desination field.*/
    if ( (dst_len = sm_n_length(dst_name)) == 0)
        return FLD_NOT_FOUND;
    if ((dst_buf=malloc(dst_len + 1)) == 0)
        return MALLOC_ERROR;

/* Call dm_expand.  */
    ret = dm_expand
          (engine, src_buf, jamtype, dst_buf, &dst_len, edit);
    if (ret == 0)
    {
        /* Write formatted text to destination field.      */
        sm_n_putfield (dst_name, dst_buf);
    }

        /* Free buffers.                        */
    free (src_buf);
    free (dst_buf);
```

```
/* If formatted string was too long for destination field,   */
/* ret will be greater than 0. If the format failed, it will */
/* be less than 0.                                           */
   return ret;
}
```

dm_getdbitext

get the text of the last executed dbms or sql command

SYNOPSIS

```
char *dm_getdbitext
```

DESCRIPTION

Use this function to get the full text of the last executed `dbms` or `sql` command. This includes all commands executed from JPL with `dbms` or `sql`, or executed from C with `dm_dbms`, `dm_dbms_noexp`, `dm_sql`, or `dm_sql_noexp`.

The text pointed to by the pointer returned by `dm_getdbitext` has a limited duration. If the application needs this information, it should call this function immediately after executing a JAM/DB*i* command. The program should process the returned string or copy it to a local variable before making additional function calls.

This is the same string that is passed to the first argument of an installed entry, error or exit handler, except that the error or exit handler is limited to 255 characters.

RETURNS

A pointer to the last executed JAM/DB*i* command

RELATED FUNCTIONS

`dbms ONERROR [JPL` ***entrypoint*** `| CALL` ***function***`]`

`dbms ONEXIT [JPL` ***entrypoint*** `| CALL` ***function***`]`

EXAMPLE

```
int
logfunc (stmt, engine, flag)
char *stmt;
char *engine;
int flag;
{
    FILE *fp;
    if (strlen(stmt) >= 255))
        stmt = dm_getdbitext();
    fp = fopen ("dbi.log", "a");
    fprintf (fp, "%s\n", stmt);
    fclose (fp);
    return 0;
}
```

dm_init

initialize JAM/DB*i* to access a specific database engine

SYNOPSIS

```
int dm_init (engine, support_routine, options, arg)
char *engine;
int support_routine;
int options;
char *arg;
```

DESCRIPTION

Before an application can access a database, JAM/DB*i* must perform an engine initialization. The initialization adds the engine name to the list of available engines, allocates a data structure for the engine, calls the engine's support routine to initialize the data structure, and sets case and error handling for the engine. Developers may use the `vendor_list` structure in `dbiinit.c` to initialize an engine at startup or else use `dm_init` to initialize an engine at a later point in the application.

The name for ***engine*** is chosen by the developer. If an application uses two or more engines, the application will use the mnemonic ***engine*** to indicate a particular DBMS. Most of the examples in the guide use a vendor name as the mnemonic, for example `sybase` or `oracle`, but any character string that is not a keyword is valid. Keywords are listed in Appendix A.. If ***engine*** is already installed, `dm_init` simply returns 0.

The name of ***support_routine*** is documented in the `dbiinit.c` file provided with the distribution. The file name is usually in the form `dm_`***vendor***`sup` where ***vendor*** is an abbreviated vendor name. Some examples are

- `dm_sybsup`
- `dm_orasup`
- `dm_intsup`

options sets some defaults for the engine. It is composed of one or two flags: ***case*** and ***error***. They may be "or-ed."

The option ***case*** sets the case-handling feature of JAM/DB*i*. It determines how JAM/DB*i* uses case to map column names to JAM variables when executing a `SELECT`. The values are

- `DM_DEFAULT_CASE` — Defaults to `DM_PRESERVE_CASE`. Another may be set by JYACC in the support routine.

- DM_PRESERVE_CASE — Use case exactly as returned by the engine.
- DM_FORCE_TO_UPPER_CASE — Force all column names returned by an engine to upper case. Therefore, the application should use upper case names for JAM variables.
- DM_FORCE_TO_LOWER_CASE — Force all column names returned by an engine to lower case. Therefore, the application should use lower case names for JAM variables.

The option ***error*** sets the behavior of the default error handler. If none is given, the default is DM_DEFAULT_DBI_MSG. The values are

- DM_DEFAULT_DBI_MSG — Restrict the default error handler to using generic JAM/DB*i* messages for all error messages.
- DM_DEFAULT_ENG_MSG — Allow the default error handler to use engine-specific messages when an error occurs.

arg is provided for future use. It should be set to 0.

Once the engine has been initialized, the application may declare a connection on it.

RETURNS

0 if there is no error,
otherwise a return code from the support routine.

RELATED FUNCTIONS

`dm_reset (`***name***`);`

EXAMPLE

```
#include "dmerror.h"
#include "smusrdbi.h"

int retcode;
retcode = dm_init( "oracle",
                   dm_orasup,
                   DM_FORCE_TO_LOWER_CASE | DM_DEFAULT_DBI_MSG,
                   0);
```

dm_reset

disable support for a named engine

SYNOPSIS

```
int dm_reset (name)
char *name;
```

DESCRIPTION

An application may call this function to disable support for a named engine.

If the routine executes successfully, it performs the following steps:

1. Closes all active connections on the engine.
2. Calls the support routine to perform any engine-specific reset processing.
3. If ***name*** was the default engine, sets the default engine to 0.
4. Frees all data structures associated with the engine.

Once an engine has been reset, the application cannot connect to the engine unless it initializes the engine with `dm_init`.

RETURNS

0 if the database engine was successfully disabled.

–1 if ***name*** was not a valid engine name.

RELATED FUNCTIONS

`dm_init (engine, support_routine, case, args);`

EXAMPLE

```
dm_reset ("oracle");
```

dm_sql

execute a SQL command directly from C

SYNOPSIS

```
int dm_sql (arg)
char *arg;
```

DESCRIPTION

Use this function to execute any SQL command directly from C.

First ***arg*** is examined for the WITH CONNECTION clause. If it is not used, dm_sql assumes the default connection. Next the colon preprocessor examines ***arg*** for colon variables. Finally, ***arg*** is passed to the appropriate routine for handing SQL commands.

After executing the requested command, JAM/DB*i* updates all global status and error variables (@dm).

If the application has installed an entry function with DBMS ONENTRY, an exit function with DBMS ONEXIT, or an error handler with DBMS ONEXIT.the installed function will be called for commands executed through the function dm_sql.

RETURNS

0 is no error,
else the return code from the default or an installed error handler

RELATED FUNCTIONS

```
int dm_dbms (arg);
```

EXAMPLE

```
int select_ssn ()
{
    int retcode;
    retcode = dm_sql ("SELECT * FROM emp WHERE ssn LIKE :+ssn");
    return retcode;
}
```

dm_sql_noexp

execute a SQL command without colon preprocessing

SYNOPSIS

```
int dm_sql_noexp (arg)
char *arg;
```

DESCRIPTION

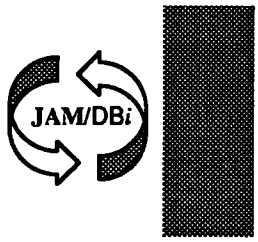
This function is identical to `dm_sql` except that colon preprocessing is NOT performed on ***arg***.

RETURNS

0 is no error,
else an code from the default or an installed error handler

RELATED FUNCTIONS

```
int dm_dbms (arg);
int dm_dbms_noexp (arg);
int dm_expand (arg);
int dm_sql (arg);
```



Chapter 16.
JAM/DB*i* *Utility Reference*

Unlike the **JAM** utilities, the **JAM/DB*i*** utilities `f2tbl` and `tbl2f` are not distributed as executables. Libraries, object code, and a makefile for `f2tbl` and `tbl2f` are included with the **JAM/DB*i*** distribution. Developers must edit the makefile to describe the environment and to supply the paths to the **JAM**, **JAM/DB*i***, and database installations.

The rest of this chapter contains reference pages for the **JAM/DB*i*** utilities:

- `f2tbl`: create a database table from a **JAM** form
- `tbl2f`: create a **JAM** form from a database table

Each reference page has the following sections:

- A synopsis of the utility, including a listing of options and arguments.
- A description of the utility.
- Examples.

f2tbl

create a database table from a JAM form

SYNOPSIS

```
f2tbl [-i] \
  [-u user [-p password]] [-s server] [-d database] [-y dictionary] \
  [-t tablename] [-l{l|u}] [-c outfile] [-f] screen ...
```

OPTIONS

`-i` Run utility in interactive mode. This opens windows where you may enter any information not supplied on the command line.

`-u` Connect with the given user name.

`-p` Connect with the given password.

`-s` Connect to the named server.

`-d` Connect to the named database.

`-y` Connect using the named dictionary.

`-t` Use ***tablename*** as the name of the database table. By default, the table name is the screen name minus `SM_FEXTENSION`.

`-l` Convert all field names to lower or upper case column names in the `CREATE` statement. For case, use `-ll` for lower or `-lu` for upper. The default is to use the case of the field names.

`-c` Write the SQL `CREATE` statement(s) to the named ASCII file. Do not create the table on the database.

`-f` Overwrite an existing database table or script file. To overwrite an existing table, `f2tbl` executes a SQL statement to drop the existing table before creating the new one. **All rows in the old table will be lost when the table is dropped.**

If no options or invalid options are given, the utility displays a usage message and a list of the valid options.

DESCRIPTION

Use this utility to create a database table or a script file for one or more **JAM** screens. If you are converting many screens, interactive mode is recommended.

For each screen, the utility defines a table with a column for each named field on the screen. The column's datatype is engine-specific and is based on the field's JAM type. If a field has a character JAM type, the utility calculates the column length by examining the field's edits. Based on the field's null field edit, the utility declares whether or not the column accepts nulls.

The `-c` flag is recommended, particularly for new users. With this flag, `f2tbl` writes the `CREATE` statement to an ASCII file where it may be examined and edited before it is executed.

Some of the logon flags are not supported on some engines. If you use an unsupported logon flag, the utility ignores it and the argument. See the engine-specific *Notes* for a list of the supported logon options.

If `f2tbl` cannot create the table, it displays either a JAM/DB*i* or engine error message.

Converting Fields to Column Definitions

COLUMN NAME

`f2tbl` uses the field name as the column name. If a field is unnamed, `f2tbl` ignores it. Please note that some valid JAM field names may be not be valid column names. For example, JAM allows the characters $ and . in JAM field names but many engines do not permit these characters in column names. If a name is illegal, `f2tbl` will display the engine's error message when it attempts to create the table.

MATCHING A JAM TYPE TO AN ENGINE DATATYPE

A field has exactly one JAM type. Since a field may have more than one of the qualifying PF4 characteristics, JAM uses precedence rules when determining the JAM type. You may determine a field's JAM type by looking at its summary screen while inside the Screen Editor.

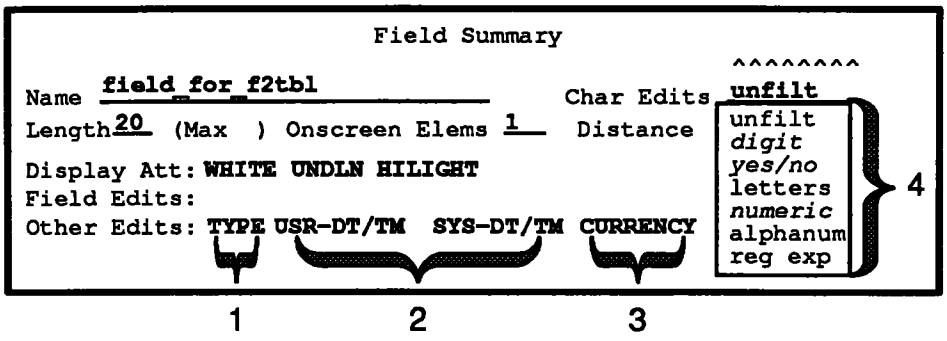


Figure 32: Field Summary Window (PF5 in draw mode). Use the summary screen to determine a field's JAM type. A TYPE edit has the highest priority, then a date time edit, then a currency edit, and finally a character edit.

Summary Keyword	Setting of Field Characteristic (PF4 menu in draw mode)	Submenu Option	JAM Type
TYPE	type (C types for structures)	char string int unsigned int short int long int float double zoned dec. packed dec.	FT_CHAR FT_INT FT_UNSIGNED FT_SHORT FT_LONG FT_FLOAT FT_DOUBLE FT_ZONED FT_PACKED
USR-DT/TM SYS-DT/TM	misc. edits	date or time	DT_DATETIME
CURRENCY	misc. edits	currency	DT_CURRENCY
Char Edits	char edits	digits only yes/no field numeric	FT_UNSIGNED DT_YESNO FT_DOUBLE

Figure 33: The keywords on the summary window indicate which of the field characteristics has set the field's JAM type.

If the word TYPE appears on the field summary window, you must press the PF4 key and choose type to open the C type submenu. The setting on the submenu indicates the JAM type. For example, if double is chosen on the submenu, the JAM type is FT_DOUBLE. Figure 33 shows the C type names and the corresponding JAM types.

If the keyword TYPE is not on the summary window, the JAM type is immediately determinable. With the keyword USR-DT/TM or SYS-DT/TM, the JAM type is DT_DATETIME. Otherwise, with the keyword CURRENCY, the JAM type is DT_CURRENCY. If none of those keywords appear, the character edit may apply: with digits only the JAM type is FT_UNSIGNED, with yes/no field the type is DT_YESNO, or with numeric the type is FT_DOUBLE.

If none of the above edits are set, but the field has a word-wrapped edit, the JAM type is FT_VARCHAR. For all other fields, the JAM type is FT_CHAR.

Since engines uses different names for datatypes, the mapping of JAM types to engine datatypes is listed in the engine-specific *Notes*.

CALCULATING THE COLUMN LENGTH

If the field has the JAM type FT_CHAR, FT_VARCHAR, or DT_YESNO, f2tbl attempts to use the field's length as the column length. For all other JAM types, a length is not calculated because the JAM type is mapped to an engine datatype with a default length.

For FT_VARCHAR fields (word-wrapped), the calculated length is the maximum shifting length times the total number of occurrences in the array. For FT_CHAR and DT_YESNO fields, the calculated length is the maximum (shifting) length of the field.

If the calculated length in either case is greater than the length permitted by the engine, f2tbl will use the maximum permitted length.

DEFINING A COLUMN AS NULL OR NOT NULL

If the field has a null field edit, the column is defined as permitting nulls. On some engines, this is the default. Others may explicitly use the keyword NULL.

If the field does not have a null field edit, the column is defined as NOT NULL.

OUTPUT

f2tbl builds a SQL CREATE statement in a form similar to the following:

```
CREATE TABLE tablename (
    column_1 datatype [(length)] [NOT] [NULL] ,
    column_2 datatype [(length)] [NOT] [NULL] ,
    ...
    column_n datatype [(length)] [NOT] NULL]
)
```

Example

Assume the screen named `inventory` has the following named fields:

- `id_no`
- `product_name`
- `price`
- `description`

The figures below show the field summary window for each field. A sample column declaration is also shown for each field. Since column datatypes are engine-specific, the names used here are solely for illustration.

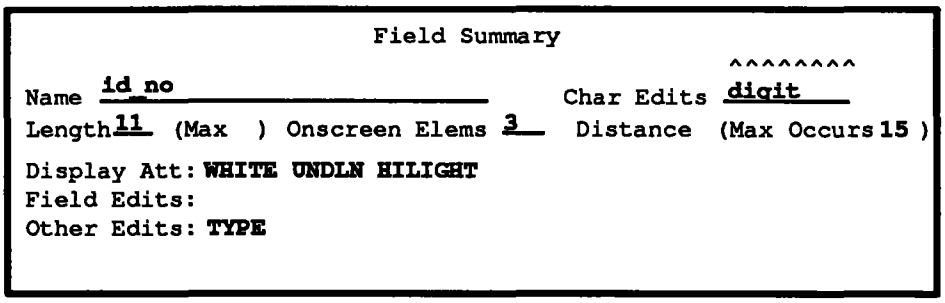


Figure 34: Field `id_no`. The type edit is set to `char string` to override the digits only character edit. Therefore, its JAM type is `FT_CHAR`.

The column definition would appear like the following

```
id_no char (11)    NOT NULL
```

Since the field does not have a word-wrapped edit, the number of occurrences is ignored. In addition, since the field does not have a null field edit, the column is defined as not allowing null values.

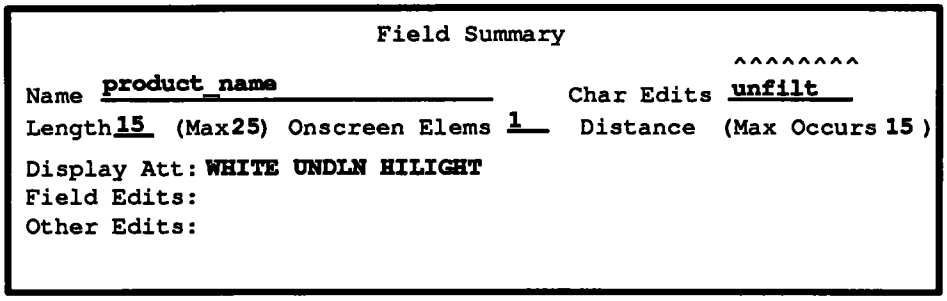


Figure 35: Field `product_name`. Its **JAM** type is `FT_CHAR`.

The column definition would appear like the following

```
product_name    char (25) NOT NULL
```

Note that the column length is 25 which is the maximum shifting length, rather than 15 which is the onscreen length. Since the field does not have a word-wrapped edit, the number of occurrences is ignored. In addition, since the field does not have a null field edit, the column is defined as not allowing null values.

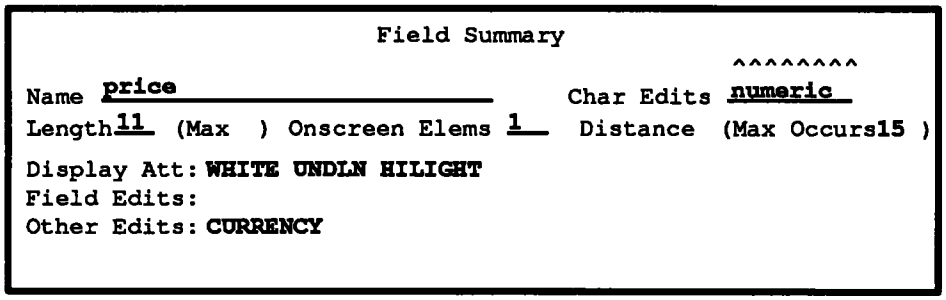


Figure 36: Field `price`. Its **JAM** type is `DT_CURRENCY`.

If the engine had a datatype called `money`, the column definition would appear like the following

```
price money     NOT NULL
```

On most engines, a currency datatype has a predefined length. In this case, `f2tbl` ignores the field's length. If the engine does not have a currency type, `f2tbl` may use a type such as `NUMERIC` or `FLOAT` and it may calculate a length or precision.

Since the field does not have a null field edit, the column is defined as not allowing null values.

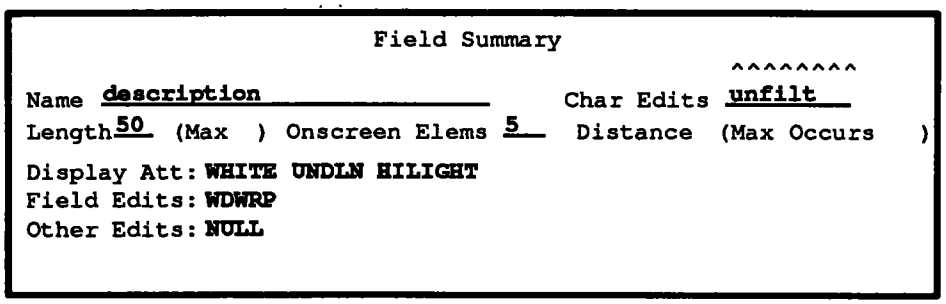


Figure 37: Field `description`. Its **JAM** type is `FT_VARCHAR`.

The column definition would appear like the following

```
description char (250)
```

Note that the column's length is the field's length 50 multiplied by the number of elements 5, and therefore 250. In this case, the field's number of occurrences affected the column length because the word-wrap edit was set. Since the field also has a null field edit, the column is defined as permitting null values. Some engines may also use the keyword NULL at the end of the definition.

The resulting CREATE statement would appear similar to the following:

```
CREATE TABLE inventory (
   id_no ( 11 ) NOT NULL,
   product_name char ( 20 ) NOT NULL,
   price money NOT NULL,
   description char ( 250 )
)
```

SEE ALSO

Engine-specific *Notes*

tbl2f

create a JAM screen from a database table

SYNOPSIS

```
tbl2f [-i] \
   [-u user [-p password]] [-s server] [-d database] [-y dictionary] \
   [-j jpl_template] [-t screen_template] \
   [-k index_key] [-l {u|l}]  [-e ext] [-f] table [table...]
```

OPTIONS

-i	Run utility in interactive mode. This opens windows where you may enter any information not supplied on the command line.
-u	Connect with the given user name.
-p	Connect with the given password.
-s	Connect to the named server.
-d	Connect to the named database.
-y	Connect using the named dictionary.
-j	Use the named file as a template for creating the JPL screen module and assigning control strings. The utility looks in the current directory and in the SMPATH directories for the named file. The default template is ***db***`exm.jpl`.
-t	Use the named file as a template for creating the JAM screen.
-k	Use the named column as the index key in the JPL procedures. If this flag is not, `tbl2f` chooses an index by querying the engine's system tables. If it cannot find one for the table, it defaults to the first column in the table.
-l	Force the case of column names in the JPL procedures and the field names on the screen to upper (`-lu`) or lower (`-ll`) case. The default is to preserve case.
-e	Append ***ext*** as the extension to the screen files. The default is SMFEXTENSION, typically JAM.
-f	Overwrite an existing screen file.

If no options or invalid options are given, the utility displays a usage message and a list of the valid options.

DESCRIPTION

Use this utility to create a JAM screen for each named database table. If you are converting many tables, interactive mode is recommended.

In each screen, `tbl2f` will create the following

- A field for each column in the table, with up to 250 fields created in total.
- Display text on the screen identifying the name of the screen and the name of each field.
- Control strings to call the JPL procedures.
- JPL procedures to query and update the table.

The following topics are covered in the remaining sections:

- Controlling the case of field names and predicting field characteristics on the created screen (page 214).
- Using a JPL template file to change and add procedures in the JPL screen module (page 216).
- Using a JPL template file to put control strings on the created screen (page 223).
- Using a screen template to change the default screen characteristics (page 225).

Fields

The utility creates a field for each column in the table, with up to 250 fields created in total. Field characteristics are assigned according to the column's data type. A field is named for its column in the table. The field's length is taken from the column definition.

FIELD NAMES

When `tbl2f` creates a field, it names the field for a database column. By default, the utility uses case exactly as returned by the database. On engines where column names are always upper case, for example ORACLE, the utility will create upper case field names by default. On engines where columns names may be in either or mixed case, the utility will create field names using the exact case of the column name.

The utility provides the option of forcing case to upper or lower. This is done with the `-l` flag on the command line or with the `Options` menu in interactive mode. Please note that this option forces the case of both onscreen field names and the column names used in the SQL statements in the JPL procedures.

To the left of each field, the utility displays the field name. Note that if the field name contains any draw field symbols, such as the underscore, those characters will be converted to fields when the screen is edited.

While almost all column names are valid JAM identifiers, `tbl2f` does not verify if a column name is a valid JAM field name and thus does not report an error for bad field names.

You may easily verify the validity of field names by using the JAM utility `f2asc` to create an ASCII version of the screen file and then run `f2asc` to convert it back to binary. Since `f2asc` validates field names before re-creating the binary file, it will report any invalid field names. If it does, you may use a text editor to quickly edit the `f2asc` ASCII file and then convert the file to a binary screen file. If the screen has JPL procedures referencing the fields, you should change only the references to the invalid field name and not change the references to the column name. For example, if the table `inventory` contained three columns `id#`, `product`, and `description`, the field names `product` and `description` are valid, but the field name `id#` is not. If the field were renamed `id_no`, a JPL statement like the following

```
sql SELECT id#, product, description FROM inventory \
   WHERE id# = :+id#
```

should be edited to

```
dbms ALIAS id# id_no
sql SELECT id#, product, description FROM inventory \
   WHERE id# = :+id_no
```

FIELD CHARACTERISTICS

When `tbl2f` creates a field, it assigns field characteristics based on the column's datatype and characteristics. The distributed JPL file ***db***`t2f.jpl`, where ***db*** is an abbreviated vendor name, equates engine datatypes with JAM types. For example, an engine datatype such as `money` is typically treated as the JAM type `DT_CURRENCY`. An engine datatype `char` is usually treated as the JAM type `FT_CHAR`. See the engine-specific *Notes* for a listing.

Based on the JAM type, the field is assigned the following edits:

Column Type Equivalent to:	Assigned Field Characteristics:		
JAM Type	C type (non-default)	misc. edits	char edits
FT_SHORT	short int		digits only
FT_INT	int		digits only
FT_UNSIGNED	unsigned int		digits only
FT_LONG	long int		digits only
FT_FLOAT	float		numeric
FT_DOUBLE	double		numeric
DT_DATETIME		date time	unfiltered
DT_CURRENCY		currency	unfiltered
FT_CHAR			unfiltered
FT_VARCHAR			unfiltered

Since engines uses different names for datatypes, the mapping of datatypes to JAM types is listed in the engine-specific *Notes*.

The length of the field depends on the field's JAM type.

- An FT_CHAR or FT_VARCHAR field is assigned the length of the column, up to the maximum length of 255.
- A DT_DATETIME column is assigned a default length of 20.
- A numeric type column is assigned an engine-specific length and precision defined in ***db***t2f.jpl.

tbl2f supports the engine's standard datatypes. Some engines permit developers to define their own datatypes. To change the JAM type of a standard datatype or to supply one for a user datatype, you must modify ***db***t2f.jpl. After editing the file, you must recompile the tbl2f executable so that the new assignments are used.

JPL Procedures

As a part of the distribution, JAM/DB*i* supplies a template of JPL procedures. It uses this template to create a JPL screen module. The default template ***db***exm.jpl builds procedures to fetch, update, insert, and delete rows in the table.

These table-specific procedures are created with the use of special tbl2f variables which begin with a double colon (::). The tbl2f variables provide strings or statements to help perform some commonly useful tasks.

There are 18 tbl2f variables. The variable names are composed of a root and a suffix. The 6-character root describes an action such as ::CLR_ for clear or ::QBEX for query by ex-

pression. The 3-character suffix describes which columns the action will involve. The roots and suffixes are described in the tables below.

Root	*Description*
`::CLR_`	for clearing the onscreen value of one or more columns in the form `cat column`
`::COND`	for a list of conditions in the form `column = :+column [:CONAND column = :+column ...]`
`::LIST`	for a list of one or more column names in the form `column [:LISTAND column ...]`
`::SET_`	for a list of one or more onscreen column values in the form `:+column [:SET_AND :+column ...]`
`::VAL_`	for a list of one or more onscreen column values in the form `:+column [:VAL_AND :+column ...]` on some engines this is equivalent to `SET_`

`::QBEX`	for `if` block(s) that build a query–by–expression clause in the form
	`if (column != "")`
	`{`
	`  CAT QBYEXAM QBEXAM VAND "column {:LIKEWORD|=} :+column "`
	`  CAT VAND LIKEAND`
	`}`

Suffix	*Description*
`ALL`	use all columns
`EIN`	use all columns except the index column
`IND`	use only the index column

Every combination of ***rootsuffix*** is a legal `tbl2f` variable.

If there any other double colon variables in the template, `tbl2f` simply strips off the first column. The utility will attempt to expand standard colon variables. If

```
:tabname
```

is used in the template, the utility replaces it with the name of the table that it is processing. If it cannot expand a colon variable it ignores it. For best results, use the backslash to preserve all variables that should be expanded by the application rather than the utility. For example,

```
# tbl2f will replace :tabname with the table name
sql SELECT * FROM :tabname

# JPL will replace :uid when the application is run
dbms DECLARE conn1 CONNECTION FOR USER \:uid
```

The sections below give an example for each root showing a suggested use in a template and its output. The output is shown in two forms, one generic and the other based on a sample table called `acc`. The table `acc` contains three columns:

- `ssn` a character column of length 11
- `salary` a money column
- `exmp` an integer column

The index column for `acc` is `ssn`.

::CLR_ VARIABLES

Use the `::CLR_` variables to create `cat` statements to clear one, some, or all the onscreen column values.

Syntax in a JPL Template

```
proc clear
::CLR_ALL
return
```

Output Syntax in a JPL Screen Module

```
proc clear
cat index_field
cat field1
cat field2
...
return
```

Output for Sample Table `acc`

```
proc clear
cat ssn
cat salary
cat exmp
return
```

::COND VARIABLES

Use a ::COND variable to get a string suitable for a WHERE clause. While all ::COND variables are legal, the condition ::CONDALL or ::CONDIND is more useful than ::CONDEIN when performing a SELECT, UPDATE, or DELETE.

If ::CONDALL is used, the conditions are separated with the JPL variable :CONDAND. In the distributed templates, CONDAND is usually the keyword AND.

Syntax in a JPL Template

```
sql SELECT * FROM :tabname WHERE ::CONDIND
```

Output Syntax in JPL Screen Module

```
sql SELECT * FROM table WHERE index_column = :+index_field
```

Output for Sample Table acc

```
sql SELECT * FROM acc WHERE ssn = :+ssn
```

::LIST VARIABLES

Use a ::LIST variable to get a string of one, some, or all column names separated by the value of the JPL variable LISTAND. In the distributed template, LISTAND is usually a comma.

Syntax in a JPL Template

```
vars LISTAND(10)
cat LISTAND ", "

sql SELECT ::LISTALL FROM :tabname
```

Output Syntax in a JPL Screen Module

```
vars LISTAND(10)
cat LISTAND ", "

sql SELECT column1 :LISTAND column2 ... FROM table
```

Output for Sample Table acc

```
vars LISTAND(10)
cat LISTAND ", "

sql SELECT ssn :LISTAND salary :LISTAND exmp FROM acc
```

::QBEX VARIABLES

Use a ::QBEX variable to create JPL statements which at runtime generate a string expression suitable for the WHERE clause of a query-by-expression procedure. For each column re-

quested by the suffix, it creates a block of statements which test if the onscreen field is empty and concatenate a JPL variable called QBYEXAM with the name of the column and its onscreen value. Other procedures may use the value of QBYEXAM as the search criteria for a SELECT or an UPDATE.

Syntax in a JPL Template

```
vars QYBEXAM LIKEWORD(10) LIKEAND(10)
cat LIKEWORD "LIKE"
cat LIKEAND "AND"

proc sellike
# Call procedure "query" to build the QBE expression
# QBYEXAM is replaced when the application is executed.
jpl query
sql SELECT * FROM :tabname \:QYBEXAM
return

proc query
# Assign a value to the JPL variable "QBYEXAM"
vars VAND(10)
cat QYBEXAM
cat VAND
# ::QBEXALL puts an "if" block for each column here:
############################################################
::QBEXALL
############################################################
if (QBYEXAM != "")
{
   cat QBEXAM " WHERE " QYEXAM
}
return 0
```

Output Syntax in a JPL Screen Module

For each FT_CHAR column, ::QBYEXAM produces the following statements:

```
if (field != "")
{
   cat QBYEXAM QBEXAM VAND "column :LIKEWORD :+field"
   cat VAND LIKEAND
}
```

For each non-FT_CHAR column (e.g. numeric and date columns), QBYEXAM produces the following statements:

```
if (field != "")
{
   cat QBYEXAM QBEXAM VAND "column = :+field"
   cat VAND LIKEAND
}
```

Output for Sample Table acc

```
vars QYBEXAM LIKEWORD(10) LIKEAND(10)
cat LIKEWORD "LIKE"
cat LIKEAND "AND"

proc sellike
# Call procedure "query" to build the QBE expression
jpl query
sql SELECT * FROM acc :QBYEXAM
return

proc query
# Assign a value to the JPL variable "QBYEXAM"
cat QYBEXAM
cat VAND
# ::QBYEXAM puts an "if" block for each column here:
#############################################################
if (ssn != "")
{
   cat QBYEXAM QYEXAM VAND " ssn :LIKEWORD :+ssn"
   cat VAND LIKEAND
}
if (salary != "")
{
   cat QBYEXAM QYEXAM VAND " salary = :+salary"
   cat VAND LIKEAND
}
if (exmp != "")
{
   cat QBYEXAM QYEXAM VAND " exmp = :+exmp"
   cat VAND LIKEAND
}
#############################################################
```

```
if (QBYEXAM != "")
{
   cat QBEXAM " WHERE " QYEXAM
}
return 0
```

::SET_ VARIABLES

Use a ::SET_ variable to get a string of the name and onscreen value of one or more columns. The pairs of column name and column value are separated by the value of the variable SET_AND. In the distributed template SET_AND is a usually comma. These variables are useful for the SET clause of an UPDATE statement.

Syntax in a JPL Template

```
vars SET_AND
cat SET_AND ","

sql UPDATE :tabname SET ::SET_EIN WHERE ...
```

Output Syntax in a JPL Screen Module

```
vars SET_AND
cat SET_AND ","

sql UPDATE table SET column1 = :+column :SET_AND \
   column2 = :+column2 ... WHERE ...
```

Output for Sample Table acc

```
vars SET_AND
cat SET_AND ","

sql UPDATE acc SET salary = :+salary :SET_AND \
   exmp = :+exmp WHERE ...
```

::VAL_ VARIABLES

Use a ::VAL_ variable to create a string of the name and onscreen value of one or more columns. The pairs of column name and column value are separated by the value of the variable VAL_AND. In the distributed template VAL_AND is a usually comma. These variables are useful for the VALUES clause of an ONSERT statement. In the distributed template, VAL_AND is a comma.

Syntax in a JPL Template

```
vars LISTAND(10) VAL_AND(10)
cat LISTAND ", "
cat VAL_AND ", "

sql INSERT INTO :tabname (::LISTALL) \
   VALUES (::VAL_ALL)
```

Output Syntax in a JPL Screen Module

```
vars LISTAND(10) VAL_AND(10)
cat LISTAND ", "
cat VAL_AND ", "

sql INSERT INTO table (column1 :LISTAND column2 ...) \
   VALUES (:+column1 :VAL_AND :+column2 ...)
```

Output for Sample Table `acc`

```
vars LISTAND(10) VAL_AND(10)
cat LISTAND ", "
cat VAL_AND ", "

sql INSERT INTO acc (ssn :LISTAND salary :LISTAND exmp) \
   VALUES (:+ssn :VAL_AND :+salary :VAL_AND :+exmp)
```

Control Strings

If a screen template is not used, control strings may be assigned to logical keys `PF1`–`PF10`, and `SPF1`–`SPF10` using the JPL template. The syntax is

```
#jctl 1      control string for PF1
#jctl 2      control string for PF2
#jctl 3      control string for PF3
#jctl 4      control string for PF4
#jctl 5      control string for PF5
#jctl 6      control string for PF6
#jctl 7      control string for PF7
#jctl 8      control string for PF8
#jctl 9      control string for PF9
#jctl 10     control string for PF10
#jctl 11     control string for SPF1
#jctl 12     control string for SPF2
#jctl 13     control string for PF13
#jctl 14     control string for PF14
#jctl 15     control string for PF15
```

```
#jctl 16     control string for PF16
#jctl 17     control string for PF17
#jctl 18     control string for PF18
#jctl 19     control string for PF19
#jctl 20     control string for SPF10
```

Note that the pound sign must be in the first column of the line and the word `jtcl` must follow it immediately. Any lines that do not begin this way are assumed to be JPL comments and they are simply copied to the JPL screen module. ***controls string*** may be any valid JAM control string. Control strings are documented in the *Author's Guide* of the JAM manual.

You may assign none, some, or all these control strings. No assignments are made for numbers outside the range of 1 to 20. The assignments may be in any order and place in the template but we recommend that you put them in a block at the beginning of the template. If the template assigns a control string more than once, the last assignment takes precedence.

In the JPL template you may wish to include a procedure that displays a status line message describing the key assignments. Remember that `%K` may be used in messages to display key-top labels. See the *JPL Guide* for more information on message display.

If a screen template is used (`-t` option), `tbl2f` ignores any control string assignments in the JPL template.

Example Template

```
#jctl 1      ^jpl select_all
#jctl 2      ^jpl select_by_index
#jctl 10     main_menu

proc message_line
msg setbkstat \
   "%KPF1: Select_All  "\
   "%KPF2: Select_by_Index  "\
   "%KPF10: Main Menu"
return

proc select_all
vars LISTAND(10)
cat LISTAND ","
sql SELECT ::LISTALL FROM :tabname
return

proc select_by_index
sql SELECT ::LISTALL FROM :tabname WHERE ::CONDIND
return
```

Screen Characteristics

An existing JAM screen may be used as a template for new screens created with `tbl2f`. A screen template is supplied with the `-t` flag or in interactive mode. If you are using a local engine on a PC, you may not have enought memory to use a screen template.

The following screen characteristics are supported by the template:

1. *Minimum number of lines and columns.* `tbl2f` will not create a screen smaller than these dimensions. If necessary, it may create a larger screen. The maximum width is the default number of columns defined in the video file. If a field is longer than the onscreen width, `f2tbl` creates a shifting field. If there are not enough onscreen lines for the fields, `tbl2f` creates a virtual screen with up to the maximum 254 lines.
2. *Border style and attribute.* `tbl2f` uses the template's border style and attribute for the new screen.
3. *Background color.* `tbl2f` uses the template's background color for the new screen.
4. *Start as menu setting.* If the template screen has menu fields, set the starting mode for the new screen.
5. *Screen-level help.* Assign a screen-level help window for the new screen.
6. *Screen entry/exit functions.* Assign screen entry and exit hook functions for the new screen.
7. *Screen-level keyset.* Assign a keyset for the screen.
8. *Display text attribute.* Use the PF4 key in draw mode to set the attributes for `pen` on the template screen. `tbl2f` will use this pen when writing labels on the new screen.

Please note that any JPL in the screen JPL module of the template is not copied to the new screen. Use the JPL template option to supply JPL procedures for the screen.

`tbl2f` has its own default attributes for the fields it creates. Any draw field symbols on the template screen are copied to the new screen, but they are not used by the utility.

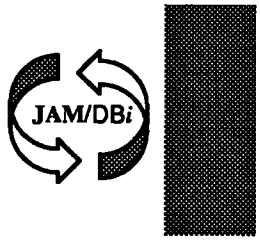
All control strings on the template screen are copied to the new screen. Any control string assignments in the JPL template are ignored.

All fields and display text on the template are written to the new screen. `tbl2f` begins writing the database fields at the first empty line below the template's display text and/or fields. The current release does not copy groups from the template.

SEE ALSO

Engine-specific *Notes*

Appendixes

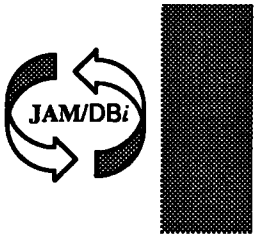


Appendix A.
Keywords

Below is a list of all the keywords for JAM/DB*i* commands. We strongly encourage developers to avoid using these keywords as identifiers, particularly for cursors, connections, engines, and transactions. We also recommend that developers avoid using these keywords when naming JAM variables which will be used in a `dbms` or `sql` statement. The list includes keywords supported by Release 4.8 and Release 5.

```
alias
autocommit

begin
binary
browse

call
cancel
catquery
checkpt_interval
close
close_all_connections
commit
connect
connected
connection
continue
continue_bottom
continue_down
continue_top
continue_up
create_proc
create_trigger
count
current

cursor
cursors

database
db
dbms
declare
disconnect
drop_proc
drop_trigger

end
engine
error
error_continue
exec
execute
execute_all

flush
file
for
format

heading

interfaces

jpl

locklevel
locktimeout
logon
logoff

max

next
null

occur
off
on
onentry
onerror
onexit
options
out
output

password
prepare_commit
print
proc
proc_control
```

```
redirect
return
retvar
rfjournal
rollback
rpc

save
schema
select
select_aliases
separator
serial

server
set
set_buffer
single_step
sql
start
stop
stop_at_fetch
store
supreps

tee
timeout
to
transaction
type

unique
update
use
user
using

warn
with
```



Appendix B.
Error and Status Codes

Like JAM, JAM/DB*i* uses symbolic constants to define its error codes. Any error handling functions written in C may simply include the header file `dmerror.h` to use these constants. JPL, on the other hand, is an interpreted language and it has no access to these constants when performing variable substitution. JPL does have access, however, to constants in the local data block (LDB). Therefore, we recommend that developers using JPL for error handling also use the data dictionary and an initialization file to define all the constants that the procedures will need. A sample data dictionary and initialization file are provided with the JAM/DB*i* distribution. Please see the `README` for directions on using these samples.

For example, if a JPL procedure must test for the no more rows signal, add the entry `DM_NO_MORE_ROWS` to the data dictionary, with length 5 and scope 1. Use an initialization file such as `const.ini` to assign its value,

```
"DM_NO_MORE_ROWS" "53256"
```

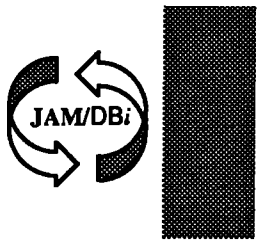
The developer may then use the name of the LDB constant in JPL procedures rather than hard-coding the decimal value in the procedure. For example, it may execute the following

```
proc select_all
sql SELECT * FROM emp
if @dmretcode == DM_NO_MORE_ROWS
   msg emsg "All rows returned."
```

The table lists the constant's name, its decimal value, and its default error message.

Constant	*Value*	*Message*
DM_NODATABASE	53249	No database selected.
DM_NOTLOGGEDON	53250	Not logged in.
DM_ALREADY_ON	53251	Already logged on.
DM_ARGS_NEEDED	53252	Arguments required.
DM_LOGON_DENIED	53253	Logon denied.
DM_BAD_ARGS	53254	Bad arguments.
DM_BAD_CMD	53255	Bad command.
DM_NO_MORE_ROWS	53256	No more rows indicator.
DM_ABORTED	53257	Processing aborted due to DB error.
DM_NO_CURSOR	53258	Cursor does not exist.
DM_MANY_CURSORS	53259	Too many cursors.
DM_KEYWORD	53260	Bad or missing keyword.
DM_INVALID_DATE	53261	Invalid date.
DM_COMMIT	53262	Commit failed.
DM_ROLLBACK	53263	Rollback failed.
DM_PARSE_ERROR	53264	SQL parse error.
DM_BIND_COUNT	53265	Incorrect number of bind vars.
DM_BIND_VAR	53266	Bad or missing bind variable.
DM_DESC_COL	53267	Describe select column error.
DM_FETCH	53268	Error during fetch.
DM_NO_NAME	53269	No name specified.

Constant	*Value*	*Message*
DM_END_OF_PROC	53270	End of procedure.
DM_NOCONNECTION	53271	No connection active.
DM_NOTSUPPORTED	53272	Command not supported for the specified engine.
DM_TRAN_PEND	53273	Transaction pending.
DM_NO_TRANSACTION	53274	Transaction does not exist.
DM_ALREADY_INIT	53275	Engine already installed.



Appendix C.
Fields in a JAM/DB*i* *Application*

JAM/DB*i* applications primarily use fields to move data between the end user and a database. Developers create a named **JAM** field for each database column that the end user will view or update.

In this chapter, we give some suggestions on creating fields for a JAM/DB*i* application. We briefly discuss how you may use the various field settings of **JAM**'s `PF4` key when creating JAM/DB*i* fields, and how these settings may affect an application. In particular, we discuss how these settings affect

- the end user's interface
- data formatting between **JAM** and a database

The physical flow of data between **JAM** and a database is discussed in detail in Chapters 8. and 9..

22.1.
JAM FIELD CHARACTERISTICS (PF4)

JAM's field characteristics provide developers with many tools for creating attractive and successful interfaces. Very briefly, we highlight here those features that are likely to be useful to JAM/DB*i* developers.

Furthermore, we discuss how the features affect data formatting between **JAM** and an engine.

22.1.1.

Field Display Attributes

The use of display attributes like color or highlight have no effect on the data.

22.1.2.

Character Edits

A character edit provides one way of helping end users enter data quickly and correctly, since it verifies each character as it is entered.

Developers may use character edits to enforce rules or checks at the application frontend. Although rules and data integrity should still be enforced by the database, effective use of character edits should reduce the number of unnecessary trips to the server, thus improving the application's efficiency.

Embedded punctuation is a useful feature with certain character edits. When a field has the character edit digits-only, letters-only, or alphanumeric the developer may save punctuation characters in the field which the user cannot type over or delete. For example, a field that accepts a U.S. telephone number would have a digits-only character edit and parentheses and a dash as embedded punctuation.

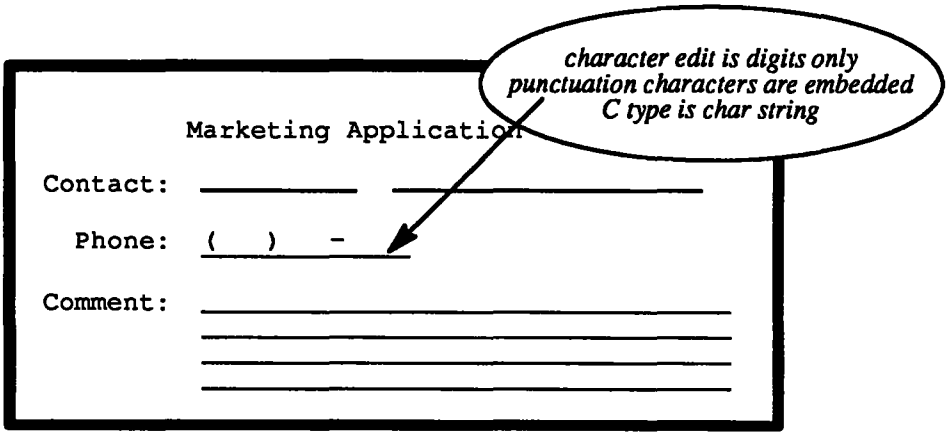


Figure 38:

JAM/DB*i* uses character edits to determine a JAM type if the field or LDB variable does **not** have any of the following edits: date/time, currency, or data type (excluding `omit` and `char string`).

Character Filter	Format Type
digits only	ft_unsigned
yes/no field	dt_yesno
numeric (+, -, .)	ft_double
all other	ft_char

22.1.3.

Field Edits

Developers may also use field edits to enforce some integrity checks at the application frontend. Remember that field edits are not enforced until the field is validated.

The field edits right justified and null field are enforced when JAM/DB*i* writes SELECT data to a field.

By default, JAM distinguishes between empty fields and null fields. To make JAM and JAM/DB*i* treat a blank field as null, you must modify the message file:

```
SM_NULLEDIT = " "
```

22.1.4.

Field Attachments

The following field attachments are useful in a JAM/DB*i* application:

- field name
- item selection
- table lookup

We discuss them below.

Field Name

This is the only required field characteristic for a JAM/DB*i* field. Database values cannot be written to unnamed fields.

Usually the developer gives a field the same name as a database column. The case of the field name is very important. In the `vendor_list` structure in `dbiinit.c` the developer sets a case flag for the engine. If the flag is DM_FORCE_TO_LOWER_CASE the developer must use lower case for the database fields. If the flag is DM_FORCE_TO_UPPER_CASE

the developer must use upper case for the database fields. If the flag is DM_PRESERVE_CASE the developer must use the exact case of the column names for the database fields.

A developer may also alias a database column to a JAM variable. This is done with the command DBMS ALIAS. When aliasing is used, the developer may use any valid JAM variable.

Item Selection and Table Lookup Screens

These attachments often improve an application's user interface. The screen entry function of the lookup or selection screen may query the database for lookup or selection values. Since the application saves the query, rather than the values, the screen maintains itself.

Developers may use the JAM library function sm_svscreen to save the selection or lookup screen in memory at runtime. If the screen is saved in memory, the application will not need to execute the query each time it displays the lookup or selection screen.

See the JAM Author's Guide and Programmer's Guide for more information.

22.1.5.
Miscellaneous Edits

Developers may execute database functions from any of the field hook functions attached in this window. Two of the miscellaneous edits may be used to format data, the date time edit and the currency edit.

JAM TYPE:

Miscellaneous Edit	Format Type	Precision
date or time field	DT_DATETIME	n/a
currency format	DT_CURRENCY	from places edit

If data is fetched to fields with either of these edits, the database values are automatically formatted with the date-time or currency edit.

22.1.6.
Field Size

The length of a field should generally be the same as the width of its associated database column. If the column is very wide, set field length to a smaller size and set the maximum

shifting length to the column width. If a field's maximum length is not equal to the width of its associated column, surplus data is truncated without warning.

Developers should set the number of elements and occurrences for a JAM/DB*i* field according to the screen size and the type of query. If a query is designed to return only one row at a time, developers should create a field with one element for each column in the row. If the query is designed to return multiple rows, the developers should create an array for each column in the row. Developers may create a scrolling array by setting the maximum number of occurrences to the greatest number of rows that may be retrieved. Developers may also create a non-scrolling array.

In brief, the two approaches are:

- Retrieve all qualifying records into large *scrolling arrays*. Each array represents a database column. The arrays usually have the same number of occurrences, so that array occurrences with the same occurrence number represent a database row. Developers may use `@dmrowcount` to ensure that the number of rows selected is less than the number of array occurrences. Users scroll through the arrays with the PgUp and PgDn keys (logical keys `SPGU` and `SPGD`). Developers may also install a customized scroll driver for an array. See the JAM Programmer's Guide for details.
- Retrieve ***n*** qualifying records incrementally into *non-scrolling arrays*. In MS-DOS environments where memory is limited, developers may wish to limit the number of rows read in at any one time. For each column, developers create an array with ***n*** non-scrolling occurrences. The first select retrieves the first ***n*** rows. Each subsequent `DBMS CONTINUE` retrieves the next ***n*** rows. To make this arrangement invisible to the user, the developers may use a key change function or a keyset to map the `DBMS CONTINUE` call to the user's physical PgDn key. Of course, the function may also be called by a standard function key. To support backward scrolling, the application may use a continuation file. A continuation file is created with the `DBMS STORE` command.

Developers may use the word-wrap edit to write long character strings to an array.

22.1.7.

Data Type

JAM data type edits have no affect on the application interface. In other words, JAM does not validate a field's contents against its data type edit and developer's cannot use this feature to perform frontend integrity checks. Developer's may use it however to set a field's format type.

When determining a variable's format type, JAM/DB*i* first checks the data type edit. If a C type is explicitly set, the keyword `TYPE` will appear on the field's summary window (`PF5` in draw mode of the JAM Screen Editor). If there is no explicit data type, or it is `omit` JAM/DB*i* will examine the variable's date-time, currency, and character edits to determine a format type. The data type edits which set format types are listed below.

Data Type	Format Type	Precision
`char string`	`FT_CHAR`	
`int`	`FT_INT`	
`unsigned int`	`FT_UNSIGNED`	
`short int`	`FT_SHORT`	
`long int`	`FT_LONG`	
`float`	`FT_FLOAT`	`yes`
`double`	`FT_DOUBLE`	`yes`
`zoned dec.`	`FT_ZONED`	
`packed dec.`	`FT_PACKED`	

Symbols

A

B

C

D

E

F

G

T

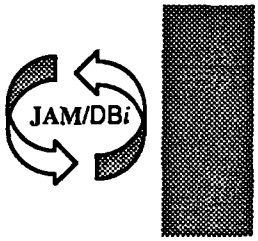
U

V

W

JAM/DB*i* for ORACLE

August 17, 1992



Notes for ORACLE

This appendix provides documentation specific to ORACLE.

It discusses the following:

- engine initialization
- connection declaration
- cursors
- formatting for colon-plus and binding
- errors and warnings
- utilities
- engine-specific features
- command directory for JAM/DB*i* ORACLE

This document is designed as a supplement, not a replacement, to the JAM/DB*i* manual. Each section identifies its companion chapter or section in the JAM/DB*i* manual.

1.1 ENGINE INITIALIZATION

See Section 7.1

By default, JAM/DB*i* uses the following values in `dbiinit.c` for ORACLE initialization:

```
static vendor_t vendor_list[] =
{
    {"oracle", dm_orasup, DM_FORCE_TO_LOWER_CASE ,(char *) 0},

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

The default settings are as follows:

`oracle`	Engine name. May be changed.
`dm_orasup`	Support routine name. Do not change.
`DM_FORCE_TO_LOWER_CASE`	Case setting for matching `SELECT` columns with JAM variable names. May be changed.

1.1.1
Engine Name and Support Routine

An application may change the engine name associated with the support routine `dm_orasup`. The application then uses that name in `DBMS ENGINE` statements and in `WITH ENGINE` clauses. For example, if you wish to use "tracking" as the engine name, make the following change:

```
static vendor_t vendor_list[] =
{
    {"tracking", dm_orasup,DM_FORCE_TO_LOWER_CASE,(char *) 0},

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

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

```
dbms ENGINE oracle_engine_name
```

where *oracle_engine_name* is the string used in `vendor_list`. For example,

```
dbms ENGINE oracle
```

or

```
dbms ENGINE tracking
```

`dm_orasup` is the name of the support routine for ORACLE. The support routine uses ORACLE's Call Interface (OCI). This name should not be changed.

If your application is using multiple engines, you need to add a line to `vendor_list` for each engine. You also need to modify your makefile to support both engines and recompile the JAM/DB*i* executables, `jxdbi` and `jamdbi`.

1.1.2
Case and Error Flags

The case flag, `DM_FORCE_TO_LOWER_CASE`, determines how JAM/DB*i* uses case when searching for JAM variables for holding `SELECT` results. JAM/DB*i* uses this setting when

comparing ORACLE column names to either a JAM variable name or to a column name in a DBMS ALIAS statement.

ORACLE is case insensitive. Regardless of the case in a SQL statement, ORACLE creates all database objects—tables, views, columns, etc.,—with upper case names. In SQL statements, users may use any case to refer to these objects. By default, JAM/DB*i* initializes case-insensitive engines using the DM_FORCE_TO_LOWER_CASE flag. This means that JAM/DB*i* attempts to match an ORACLE column name to a lower case JAM variable name when processing SELECT results. If your application is using this default, use lower case names when creating JAM variables.

The case setting may be changed. If you wish to use upper case JAM variable names, replace DM_FORCE_TO_LOWER_CASE with DM_PRESERVE_CASE or DM_FORCE_TO_UPPER_CASE.

You may also set an optional flag to change the behavior of JAM/DB*i*'s default error handler. An application may set either of the following:

DM_DEFAULT_DBI_MSG	Set the default error handler to display standard JAM/DB*i* messages for all error messages.
DM_DEFAULT_ENG_MSG	Set the default error handler to display ORACLE error messages instead of JAM/DB*i* error messages.

If neither flag is used, DM_DEFAULT_DBI_MSG is the default. To show ORACLE error messages as the default, use the bitwise OR operator and DM_DEFAULT_ENG_MSG :

```
static vendor_t vendor_list[] =
{
    {"oracle", dm_orasup, DM_FORCE_TO_LOWER_CASE | DM_DEFAULT_ENG_MSG,
    (char *) 0 },

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

If you modify the settings in `dbiinit.c`, you must recompile and link the JAM/DB*i* executables, `jxdbi` and `jamdbi`. `dbiinit.c` does not affect the utility executables, `tbl2f` and `f2tbl`.

Please note that DM_DEFAULT_DBI_MSG and DM_DEFAULT_ENG_MSG do not affect an application using an error hook function. An error hook function is installed with DBMS ONERROR and controls all error message display.

1.2 CONNECTION

See Section 7.2

The following options are supported for connections to ORACLE:

USER	***user_name***
PASSWORD	***password***

where ***user_name*** is a valid logon name for the ORACLE database. ***user_name*** must have CONNECT privileges to logon. For more information see your DBA or the ORACLE RDBMS Database Administrator's Guide.

The syntax is,

```
dbms [WITH ENGINE engine] DECLARE connection CONNECTION \
   [FOR USER user_name [PASSWORD password] ]
```

For example,

```
dbms DECLARE dbi_session CONNECTION FOR \
   USER :+uname PASSWORD :+pword
```

where `uname` and `pword` are JAM field names.

ORACLE allows your application to use one or more connections. The application may declare any number of named connections with `DBMS DECLARE CONNECTION` statements.

1.3 CURSORS

See Section 7.3

JAM/DB*i* uses two cursors for operations performed by `sql` and its equivalents, `dm_sql` and `dm_sql_noexp`. In ORACLE terminology, a cursor is also known as a *context area*. JAM/DB*i* uses one cursor for `SELECT` statements and the other for non-`SELECT` statements. These two cursors may be sufficient for small applications. Larger applications often require more; an application may declare named cursors using `DBMS DECLARE CURSOR`. For example, master and detail applications often need to declare at least one named cursor: one cursor selects the master rows and additional cursors select detail rows. In short, if an application is processing a `SELECT` set in increments (i.e., by using `DBMS CONTINUE`) while it is executing other `SELECT` statements, two or more cursors are necessary.

Declaring a named cursor may improve the performance of some `SELECT` statements. In particular, if an application is executing a `SELECT` statement more than once and the `SELECT` fetches 40 or more columns from a remote server, a named cursor is recommended. In this

case, the parse and describe is done just once when the cursor is declared, not each time the cursor is executed.

JAM/DB*i* does not put any limit on the number of cursors an application may declare to an ORACLE engine. Since each cursor requires memory and ORACLE resources, however, it is recommended that applications close a cursor when it is no longer needed.

1.4 FORMATTING FOR COLON-PLUS AND BINDING

See Chapter 8

JAM/DB*i* uses ORACLE's built-in TO_DATE function and the ORACLE format string, `ddmmyyyy hh24miss` to convert JAM dates to ORACLE form.

1.5 SCROLLING

See Section 9.1.2

ORACLE does not have native support for backward scrolling in a SELECT set. Before using any of the following commands

```
dbms [WITH CURSOR cursor] CONTINUE_BOTTOM
dbms [WITH CURSOR cursor] CONTINUE_TOP
dbms [WITH CURSOR cursor] CONTINUE_UP
```

the application must set up a continuation file for the cursor. This is done with the command

```
dbms [WITH CURSOR cursor] STORE FILE [filename]
```

1.6 ERROR AND STATUS INFORMATION

See Section 9.2 and Chapter 13

In Release 5, JAM/DB*i* uses the global variables described in the following sections to supply error and status information in an application. Note that some global variables may not be used in the current release; however, these variables are reserved for use in other engines and for use in future releases of JAM/DB*i* for ORACLE.

1.6.1
Errors

JAM/DB*i* initializes the following global variables for error code information:

`@dmretcode`	Standard JAM/DB*i* status code.
`@dmretmsg`	Standard JAM/DB*i* status message.
`@dmengerrcode`	ORACLE error code.
`@dmengerrmsg`	ORACLE error message.
`@dmengreturn`	Not used in JAM/DB*i* for ORACLE.

ORACLE returns error codes and messages when it aborts a command. It aborts a command usually because the application used an invalid option or because the user did not have the authority required for an operation. JAM/DB*i* writes ORACLE error codes to the global variable `@dmengerrcode` and writes ORACLE messages to `@dmengerrmsg`.

All ORACLE errors are JAM/DB*i* errors. Therefore, JAM/DB*i* always calls the default or the installed error handler when an error occurs.

The easiest way to test for ORACLE errors is with an installed error or exit handler. For example,

```
dbms ONERROR JPL errors
dbms DECLARE dbi_session CONNECTION FOR ...

proc errors
parms stmt engine flag
   if @dmengerrcode == 0
      msg emsg "JAM/DBi error: " @dmretmsg
   else
      msg emsg "JAM/DBi error: " @dmretmsg " %N" \
      ":engine error is" @dmengerrcode " " @dmengerrmsg
   return 1
```

If you need additional information about ORACLE errors, please consult your ORACLE documentation.

1.6.2
Warnings

JAM/DB*i* initializes the following global variables for warning information:

`@dmengwarncode`	ORACLE bit warning flag.
`@dmengwarnmsg`	Not used in JAM/DB*i* for ORACLE.

ORACLE uses a warning byte called `flags1` to signal conditions it considers unusual but not fatal. `@dmengwarncode` derives its value from this byte. `@dmengwarncode` is an 8–occurrence array. If ORACLE sets a bit in `flags1` of the Cursor Data Area, JAM/DB*i* puts a "W" in the corresponding occurrence of `@dmengwarncode`. The settings for `flags1` in ORACLE 6.0 are:

Bit Value	*Meaning*
001	There is a warning. This is set when any other bit in `flags1` is set.
002	Set if any data item was truncated.
003	Unused.
004	Unused
005	Set if an `UPDATE` or `DELETE` statement does not contain a `WHERE` clause.
006	Unused.
007	Unused.
008	Unused.

Before using `@dmengwarncode`, you should verify these settings by consulting your *Oracle Call Interfaces Manual*.

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

```
proc check_status
parms stmt engine flag

if @dmretcode == 0
{
    if @dmengwarncode [1] == "W"
    {
        if @dmengwarncode [2] == "W"
            msg emsg "Some data was truncated."
```

```
        else if @dmengwarncode [5] == "W"
           msg emsg "The operation did not use a WHERE clause."
    }
}
return
```

1.6.3
Row Information

JAM/DB*i* initializes the following global variables for row information:

`@dmrowcount`	Count of the number of ORACLE rows affected by an operation.
`@dmserial`	Not used in JAM/DB*i* for ORACLE.

ORACLE returns a count of the rows affected by an operation. JAM/DB*i* writes this value to the global variable `@dmrowcount`.

As explained on the manual page for `@dmrowcount`, the value of `@dmrowcount` after a `SELECT` is the number of rows fetched to JAM variables which may be less than or equal to the total number of rows in the select set. Immediately after an `INSERT`, `UPDATE`, or `DELETE`, `@dmrowcount` is set to the total number of rows affected by the operation. This variable is cleared whenever a `DBMS COMMIT` statement is executed.

1.7
UTILITIES

See Chapter 16

If you start the utilities in interactive mode using the `-i` flag, the utility displays an engine-independent logon screen. JAM/DB*i* uses the following options:

- `User`
- `Password`

when declaring a connection to ORACLE for the utilities. Enter the same information you use to declare a connection in `jamdbi`. The other fields on the logon screen may remain empty.

1.7.1
f2tbl

`f2tbl` creates a database table based on a JAM form. It uses each named field on the form to create a column, translating field edits to an appropriate ORACLE column definition. The table below shows the default ORACLE column definitions for each JAM type.

If you do not know how to check a field's JAM type, please see the *Utility Reference Chapter* of the JAM/DB*i* manual.

JAM *Type*	*ORACLE Column Definition*		
	Type	*Length*	*Precision*
DT_CURRENCY	NUMBER	Same as field length (maximum of 42)	Field's precision[1]
DT_DATETIME	DATE		
DT_YESNO	CHAR	Same as field length (maximum of 240)	
FT_CHAR	CHAR	Same as field length (maximum of 240)	
FT_DOUBLE	NUMBER	Same as field length (maximum of 42)	Field's precision[1]
FT_FLOAT	NUMBER	Same as field length (maximum of 42)	Field's precision[1]
FT_INT	NUMBER	Same as field length (maximum of 42)	0
FT_LONG	NUMBER	Same as field length (maximum of 42)	0
FT_PACKED	NUMBER	Same as field length (maximum of 42)	Field's precision[1]
FT_SHORT	NUMBER	Same as field length (maximum of 42)	0
FT_UNSIGNED	NUMBER	Same as field length (maximum of 42)	0
FT_VARCHAR	LONG	Same as field length	
FT_ZONED	NUMBER	Same as field length (maximum of 42)	Field's precision

1. If the field length is greater than 42, the precision is adjusted using the calculation: precision – field length + 42

To change these defaults you must edit the JPL procedure `type` in the distribution JPL module `oraf2t.jpl`, compile it by using `jpl2bin`, and replace the previous version in `orajpl.lib` by using `formlib -r`.

1.7.2

tbl2f

`tbl2f` creates a JAM form based on an ORACLE table. It creates a field for each column in the table, using the column's datatype to assign the appropriate field characteristics. The table below lists the default field lengths and precisions for each ORACLE datatype. It also lists a default JAM type.

Although ORACLE columns names are upper case, JAM/DB*i* by default uses lower case when creating field names for a `tbl2f` screen. This is consistent with the default case setting for ORACLE in `dbiinit.c` (see Section 1.1). If you changed the default in `dbiinit.c` to `DM_PRESERVE_CASE` or `DM_FORCE_TO_UPPER_CASE`, you should set the case option of `tbl2f` to match. The case option may be set on the command line or from a pull-down menu in interactive mode. For example, to start `tbl2f` in interactive mode and use upper case for JAM variables, type

```
tbl2f -i -lu
```

Note that there are additional characteristics associated with each JAM type. Those are described in the *Utility Reference Chapter* of the JAM/DB*i* manual.

ORACLE Type	JAM *Field Definition*		
	JAM *Type*	*Length*	*Precision*
NUMBER(0 length)	FT_FLOAT	16	5
NUMBER (0 precision)	FT_INT	Same as column length	
CHAR	FT_CHAR	Same as column length	
DATE	DT_DATETIME	20	
LONG	FT_VARCHAR	Same as column length (maximum of 255)	

To change these defaults, you must edit the JPL procedure `type` in the distribution JPL module `orat2f.jpl`, compile it by using `jpl2bin`, and replace the previous version in `orajpl.lib` by using `formlib -r`.

1.8 ORACLE-SPECIFIC COMMANDS

See Chapter 11

JAM/DB*i* for ORACLE provides additional commands for ORACLE-specific features. If you are using multiple engines or are porting an application to or from another engine, please note that these commands may work differently or may not be supported on some engines.

1.8.1 Using Transactions

JAM/DB*i* supports the following commands when using transactions. See the reference pages for more information on each command.

AUTOCOMMIT	turn on or off autocommit processing
COMMIT	commit a transaction
ROLLBACK	rollback a transaction

ORACLE transactions are per connection. An application must test for errors during the transaction and terminate a transaction by issuing an explicit ROLLBACK or COMMIT.

When an application closes a connection with CLOSE_ALL_CONNECTIONS or CLOSE CONNECTION, ORACLE commits any pending transactions on those connections. If an application terminates without explicitly closing its connections, ORACLE rolls back any pending transactions on those connections. However, these procedures are not recommended. Instead, it is strongly recommended that applications use explicit COMMIT and ROLLBACK statements to terminate transactions.

AUTOCOMMIT

turn autocommit on or off

SYNOPSIS

```
dbms [WITH CONNECTION connection] AUTOCOMMIT ON
dbms [WITH CONNECTION connection] AUTOCOMMIT OFF
```

DESCRIPTION

This command controls whether changes to a database occur immediately upon execution of a SELECT, INSERT, UPDATE, or DELETE command, or whether they occur when a DBMS COMMIT is explicitly executed.

If the WITH CONNECTION clause is not used, JAM/DB*i* applies the AUTOCOMMIT setting to the application's default connection.

The default mode is AUTOCOMMIT OFF. This means that the engine automatically starts a transaction after an application declares a connection. When a recoverable statement (INSERT, UPDATE, and DELETE) is executed, it is not automatically committed. The effects of the statement are not visible until the transaction is terminated. If the transaction is terminated by DBMS COMMIT, the updates are committed and visible to other users. If the transaction is terminated by DBMS ROLLBACK, the updates are not committed, and the database is restored to its state prior to the start of the transaction. Once a transaction is terminated, the engine automatically begins a new transaction.

Developers may change the default behavior by using the AUTOCOMMIT ON mode. In this mode, a statement is committed automatically upon successful execution. Its effects are immediately visible to other users, and it cannot be rolled back.

ORACLE recommends AUTOCOMMIT OFF mode because it may improve performance.

In AUTOCOMMIT OFF mode, an application should issue a COMMIT at the end of each logical unit of work. It should also use an error handler to test for errors and perform rollbacks as needed.

RELATED FUNCTIONS

```
dbms [WITH CONNECTION connection] COMMIT

dbms [WITH CONNECTION connection] ROLLBACK
```

EXAMPLE

```
proc tran_handle
   vars jpl_retcode
   retvar jpl_retcode
   dbms WITH CONNECTION xxx1 AUTOCOMMIT OFF
   jpl update_emp
# If the procedure "update_emp" executes successfully,
# jpl_retcode = 0; if a statement fails, jpl_retcode = -1
# or the value returned by the installed error handler.
# For all errors, execute a ROLLBACK.
   if jpl_retcode
   {
      dbms ROLLBACK
      msg emsg "New employee data NOT entered."
   }
   else
      msg emsg "New employee data successfully entered."
return 0

proc new_emp
   sql INSERT INTO emp \
   (ssn, last, first, street, city, st, zip, grade) VALUES
      (:+ssn, :+last, :+first, \
      :+street, :+city, :+st, :+zip, :+grade)
   sql INSERT INTO review (ssn, revdate, newsal, newgrd) \
      VALUES (:+ssn, :+hiredate, :+sal, :+grd)
   sql INSERT INTO acc (ssn, sal, exmp) \
      VALUES (:+ssn, :+sal, :+exmp)
   dbms COMMIT
   return 0
```

COMMIT

commit a transaction

SYNOPSIS

```
dbms [WITH CONNECTION connection] COMMIT
```

DESCRIPTION

Use this command to commit a pending transaction. Committing a transaction saves all the work since the last COMMIT. Changes made by the transaction become visible to other users. If the transaction is terminated by DBMS ROLLBACK, the updates are not committed, and the database is restored to its state prior to the start of the transaction. Once a transaction is terminated, the engine automatically begins a new transaction.

If the WITH CONNECTION clause is not used, JAM/DB*i* issues the commit on the default engine.

Before beginning a transaction, the application should ensure that the connection is using AUTOCOMMIT OFF mode; this is usually the default. It should COMMIT or ROLLBACK any pending transactions before starting a new one.

If an application is using AUTOCOMMIT ON mode, this command is not needed.

RELATED FUNCTIONS

```
dbms [WITH CONNECTION connection] AUTOCOMMIT {ON | OFF}
dbms [WITH CONNECTION connection] ROLLBACK
```

EXAMPLE

Refer to the example shown for AUTOCOMMIT.

ROLLBACK

rollback a transaction

SYNOPSIS

```
dbms [WITH CONNECTION connection] ROLLBACK
```

DESCRIPTION

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

If the `WITH CONNECTION` clause is not used, JAM/DB*i* issues the rollback on the default engine.

If a statement in a transaction fails, an application must attempt to reissue the statement successfully or else rollback the transaction. If an application cannot complete a transaction, it should rollback the transaction. If it does not, it may inadvertently commit the partial transaction when it commits a later transaction.

RELATED FUNCTIONS

```
dbms [WITH CONNECTION connection] AUTOCOMMIT {ON | OFF}
dbms [WITH CONNECTION connection] COMMIT
```

EXAMPLE

Refer to the example shown for `AUTOCOMMIT`.

1.9
COMMAND DIRECTORY FOR ORACLE

This section contains a directory for all the commands available in JAM/DB*i* for ORACLE. The following table lists the command, a short description of the command, and the location of the reference page for that command. If the location is described as ORACLE Notes, that information is enclosed in this document.

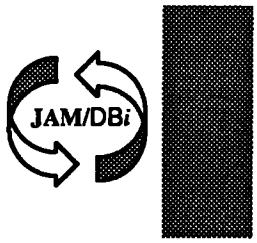
Command	*Description*	*Documentation*
ALIAS	name a JAM variable as the destination of a selected column or aggregate function	JAM/DB*i* Manual
AUTOCOMMIT	turn on or off autocommit processing	ORACLE Notes
BINARY	create a JAM/DB*i* variable for fetching binary values	JAM/DB*i* Manual
CATQUERY	redirect SELECT results to a file or a JAM variable	JAM/DB*i* Manual
CLOSE CONNECTION	close a named connection	JAM/DB*i* Manual
CLOSE CURSOR	close a cursor	JAM/DB*i* Manual
CLOSE_ALL_CONNECTIONS	close all connections on all engines	JAM/DB*i* Manual
COMMIT	commit a transaction	ORACLE Notes
CONNECTION	set a default connection and engine for the application	JAM/DB*i* Manual
CONTINUE	fetch the next screenful of rows from a SELECT set	JAM/DB*i* Manual
CONTINUE_BOTTOM	fetch the last screenful of rows from a SELECT set	JAM/DB*i* Manual

Command	*Description*	*Documentation*
CONTINUE_DOWN	fetch the next screenful of rows from a SELECT set	JAM/DB*i* Manual
CONTINUE_UP	fetch the previous screenful of rows from a SELECT set	JAM/DB*i* Manual
CONTINUE_TOP	fetch the first screenful of rows from a SELECT set	JAM/DB*i* Manual
DECLARE CONNECTION	declare a named connection to an engine	JAM/DB*i* Manual
DECLARE CURSOR	declare a named cursor	JAM/DB*i* Manual
ENGINE	set the default engine for the application	JAM/DB*i* Manual
EXECUTE	execute a named cursor	JAM/DB*i* Manual
FORMAT	format the results of a CATQUERY	JAM/DB*i* Manual
OCCUR	set the number of rows for JAM/DB*i* to fetch to an array and choose an occurrence where JAM/DB*i* should begin writing result rows	JAM/DB*i* Manual
ONENTRY	install a JPL procedure or C function which JAM/DB*i* will call before executing a sql or dbms statement	JAM/DB*i* Manual
ONERROR	install a JPL procedure or C function which JAM/DB*i* will call whenever a sql or dbms statement fails	JAM/DB*i* Manual
ONEXIT	install a JPL procedure or C function which JAM/DB*i* will call after executing a sql or dbms statement	JAM/DB*i* Manual

Command	*Description*	*Documentation*
`ROLLBACK`	rollback a transaction	ORACLE Notes
`START`	set the first row for JAM/DB*i* to return from a `SELECT` set	JAM/DB*i* Manual
`STORE FILE`	store the rows of a `SELECT` set in a temporary file so that the application may scroll through the rows	JAM/DB*i* Manual
`UNIQUE`	suppress repeating values in a selected column	JAM/DB*i* Manual
`WITH CONNECTION`	set the default connection for the duration of a command	JAM/DB*i* Manual
`WITH CURSOR`	specify the cursor to use for a statement	JAM/DB*i* Manual
`WITH ENGINE`	set the default engine for the duration of a command	JAM/DB*i* Manual

JAM/DB*i* for SYBASE

August 19, 1992



Notes for SYBASE

This appendix provides documentation specific to SYBASE.

It discusses the following:

- engine initialization
- connection declaration
- cursors
- formatting for colon-plus and binding
- errors and warnings
- utilities
- engine-specific features
- command directory for JAM/DB*i* SYBASE

This document is designed as a supplement, not a replacement, to the JAM/DB*i* manual. Each section identifies its companion chapter or section in the JAM/DB*i* manual.

1.1 ENGINE INITIALIZATION *See* JAM/DB*i* *Manual – Section 7.1*

By default, JAM/DB*i* uses the following values in `dbiinit.c` for SYBASE initialization:

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

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

The default settings are as follows:

`sybase`	Engine name. May be changed.
`dm_sybsup`	Support routine name. Do not change.
`DM_PRESERVE_CASE`	Case setting for matching `SELECT` columns with JAM variable names. May be changed.

1.1.1

Engine Name and Support Routine

An application may change the engine name associated with the support routine `dm_sybsup`. The application then uses that name in `DBMS ENGINE` statements and in `WITH ENGINE` clauses. For example, if you wish to use "tracking" as the engine name, make the following change:

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

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

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

```
dbms ENGINE sybase_engine_name
```

where ***sybase_engine_name*** is the string used in `vendor_list`. For example,

```
dbms ENGINE sybase
```

or

```
dbms ENGINE tracking
```

`dm_sybsup` is the name of the support routine for SYBASE. This name should not be changed.

If your application is using multiple engines, you need to add a line to `vendor_list` for each engine. You also need to modify your makefile to support both engines and recompile the JAM/DB*i* executables, `jxdbi` and `jamdbi`.

1.1.2

Case and Error Flags

The case flag, `DM_PRESERVE_CASE`, determines how JAM/DB*i* uses case when searching for JAM variables for holding `SELECT` results. JAM/DB*i* uses this setting when

comparing SYBASE column names to either a JAM variable name or to a column name in a DBMS ALIAS statement.

SYBASE is case-sensitive. SYBASE uses the exact case of a SQL statement when creating database objects like tables and columns. In a SQL statement, users must use the same exact case when referring to these objects. By default, JAM/DB*i* initializes case-sensitive engines using the DM_PRESERVE_CASE flag. This means that JAM/DB*i* matches the SYBASE column name to a JAM variable with the same name and case.

By changing this flag, you can force JAM/DB*i* to perform case-insensitive searches. Use DB_FORCE_TO_LOWER_CASE to match SYBASE column names to lower case JAM names; use DM_FORCE_TO_UPPER_CASE to match to upper case JAM names.

You may also set an optional flag to change the behavior of JAM/DB*i*'s default error handler. An application may set either of the following:

DM_DEFAULT_DBI_MSG	Set the default error handler to display standard JAM/DB*i* messages for all error messages.
DM_DEFAULT_ENG_MSG	Set the default error handler to display SYBASE error messages instead of JAM/DB*i* error messages.

If neither flag is used, DM_DEFAULT_DBI_MSG is the default. To show SYBASE error messages as the default, use the bitwise OR operator and DM_DEFAULT_ENG_MSG:

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

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

If you modify the settings in `dbiinit.c`, you must recompile and link the JAM/DB*i* executables, `jxdbi` and `jamdbi`. `dbiinit.c` does not affect the utility executables, `tbl2f` and `f2tbl`.

Please note that DM_DEFAULT_DBI_MSG and DM_DEFAULT_ENG_MSG do not affect an application using an error hook function. An error hook function is installed with DBMS ONERROR and controls all error message display.

1.2 CONNECTION

See JAM/DB*i* *Manual – Section 7.2*

The following options are supported for connections to SYBASE:

USER	***user_name***
PASSWORD	***password***
SERVER	***server_name***
DATABASE	***database_name***
INTERFACES	***interfaces_file_pathname***
CURSORS	***1 | 2***
TIMEOUT	***seconds***

Use INTERFACES to supply the pathname to an interfaces file. An interfaces file contains the name and network address of every SYBASE server available on the network. If this option is not used, SYBASE looks for a file called `interfaces` in the SYBASE parent directory (e.g., `/usr/sybase/interfaces`). This option is ignored for OS/2, MS-DOS, and Windows applications.

Use TIMEOUT to set the number of seconds that Open Client waits for a SYBASE response to a request for a connection. A timeout of 0 seconds represents an infinite timeout period. The default is usually 60 seconds.

Use CURSORS to control the number of default cursors JAM/DB*i* creates when the application declares a connection. The default is 1. This means that JAM/DB*i* uses one cursor for any operation executed with `sql` or `dm_sql`, whether it is a SELECT or non-SELECT operation. The application must set CURSORS to 2 to use browse mode. You may also wish to use two default cursors if your application switches between a SELECT and non-SELECT operations. See the section on cursors for additional information.

The syntax for declaring a connection is,

```
dbms DECLARE connection CONNECTION FOR \
   USER user_name PASSWORD password DATABASE database \
   SERVER server INTERFACES interface_pathname \
   TIMEOUT timeout CURSORS number_of_cursors
```

For example,

```
dbms DECLARE dbi_session CONNECTION FOR \
   USER :uname PASSWORD :pword DATABASE sales \
   SERVER birch INTERFACES '/usr/sybase/interfaces.app'
   TIMEOUT 15 CURSORS 2
```

where `uname` and `pword` are JAM field names.

SYBASE allows your application to use one or more connections. The application may declare any number of named connections with DBMS DECLARE CONNECTION statements, up to the maximum number permitted by the server.

1.3 CURSORS

See JAM/DB*i* *Manual – Section 7.3*

JAM/DB*i* uses two cursors for operations performed by `sql` and its equivalents, `dm_sql` and `dm_sql_noexp`. JAM/DB*i* uses one cursor for `SELECT` statements and the other for non-`SELECT` statements. These two cursors may be sufficient for small applications. Larger applications often require more; an application may declare named cursors using `DBMS DECLARE CURSOR`. For example, master and detail applications often need to declare at least one named cursor: one cursor selects the master rows and additional cursors select detail rows. In short, if an application is processing a `SELECT` set in increments (i.e., by using `DBMS CONTINUE`) while it is executing other `SELECT` statements, two or more cursors are necessary.

JAM/DB*i* does not put any limit on the number of cursors an application may declare to an SYBASE engine. Since each cursor requires memory and SYBASE resources, however, it is recommended that applications close a cursor when it is no longer needed.

1.4 FORMATTING FOR COLON-PLUS AND BINDING

See JAM/DB*i* *Manual – Chapter 8*

SYBASE requires a leading dollar sign for values inserted in a `money` column in order to ensure precision. JAM/DB*i* will use a leading dollar sign when it formats `DT_CURRENCY` values. Any other amount formatting characters are stripped. Therefore, if a currency field contained

```
500,000.00
```

JAM/DB*i* would format it as

```
$500000.00
```

1.5 SCROLLING

See JAM/DB*i* *Manual – Section 9.1.2*

SYBASE has native support for backward scrolling in a `SELECT` set. Before using any of the following commands

```
dbms [WITH CURSOR cursor] CONTINUE_BOTTOM
dbms [WITH CURSOR cursor] CONTINUE_TOP
dbms [WITH CURSOR cursor] CONTINUE_UP
```

the application must specify whether to use native scrolling or JAM/DB*i* scrolling. To use native scrolling, use the command

```
dbms [WITH CURSOR cursor] SET_BUFFER arg
```

where *arg* is the number of rows to buffer.

To use JAM/DB*i* scrolling, use the command

```
dbms [WITH CURSOR cursor] STORE FILE [filename]
```

1.5.1

Locking

JAM/DB*i* SYBASE developers should consider locking issues when building applications that `SELECT` large amounts of data.

When an application executes a `SELECT` that returns many rows, SYBASE may use a "shared lock" to preserve read-consistency. That is, to preserve the state of the selected data, SYBASE may prevent other applications or users from changing the data until the application has received all the rows. This behavior is usually seen for `SELECT` sets that contain 500 or more rows.

As a part of developing and testing an application, JAM/DB*i* developers should monitor SYBASE's behavior by running the SYBASE command `sp_lock` from another terminal when the application executes a `SELECT`. If a `SELECT` executed by a JAM/DB*i* application is holding a lock, the cursor's *spid* will be listed.

Since a shared lock prevents other users from updating data, it is important to release shared locks as soon as possible. To release a shared locked,

- get all the rows in the `SELECT` set, or
- flush pending rows in the `SELECT` set

An application has two ways of getting the entire `SELECT` set:

- create JAM arrays which are large enough to hold the entire `SELECT` set, or
- use `DBMS STORE FILE` and `DBMS CONTINUE_BOTTOM` to buffer all the rows in a temporary file on disk

For example, an application may set up a continuation file before executing a SELECT. Before returning control to the user, the application may execute DBMS CONTINUE_BOTTOM which forces JAM/DB*i* get all the rows from the SELECT set and buffer them in a temporary file. This also forces SYBASE to release any shared lock it is holding for the SELECT.

In the following example, the application puts a message on the status line and flushes the display. Next is sets up a continuation file and executes the SELECT. It calls CONTINUE_BOTTOM to force JAM/DB*i* to get all the rows. Finally, it calls CONTINUE_TOP to ensure that the SELECT set's first page (rather than its last page) of rows is displayed when control is returned to the user.

```
proc big_select
   msg setbkstat "Processing. Please be patient..."
   flush
   dbms STORE FILE
   sql SELECT ....
   dbms CONTINUE_BOTTOM
   dbms CONTINUE_TOP
return
```

An application may also limit the number of rows a use may view at a time by using the DBMS FLUSH command. When this command is executed, SYBASE discards any pending rows and releases all associated locks. For example,

```
proc big_select
   sql SELECT ....
   if @dmretcode != DM_NO_MORE_ROWS
      dbms FLUSH
return
```

To monitor lock information within the application, the application may query SYBASE for the spid number of a cursor and the number of locks held by the cursor. Note that each cursor has its own spid and it keeps the same spid number until the application closes the cursor. To get a cursor's spid number, an application must use the cursor to select the global SYBASE variable @@spid.

```
# Get the SYBASE spid for a JAM/DBi cursor
# before SELECTing rows.
proc get_spid
parms cursor
vars spid
   if cursor == ""
      sql SELECT spid = @@spid
   else
```

```
    {
       dbms DECLARE  :cursor CURSOR FOR \
          SELECT spid = @@spid
       dbms EXECUTE :cursor
    }
    return spid

# Get the number of locks held by a SYBASE spid.
proc lockstatus
    parms spid4select
    vars lcount
    dbms DECLARE lock_count CURSOR FOR \
       SELECT COUNT(*) FROM master.dbo.syslocks \
       WHERE spid = :spid4select
    dbms WITH CURSOR lock_cursor ALIAS lcount
    dbms WITH CURSOR lock_cursor EXECUTE
    dbms CLOSE CURSOR lock_cursor
    return lcount
```

An application may get a cursor's spid before executing a SELECT for rows. After fetching rows the application may query SYBASE for the number of locks. Note that the order of these statements is important: if an application attempts to get a cursor's spid *after* fetching rows, the SELECT for the cursor's spid will release any locks and any pending rows. For this reason, be sure to get the cursor's spid **before** fetching rows. See the example below.

```
proc select
vars cursor_spid locks

    retvar cursor_spid
    jpl get_spid "c1"
    retvar

    dbms DECLARE c1 CURSOR FOR SELECT ...
    dbms WITH CURSOR c1 EXECUTE

    retvar locks
    jpl lockstatus :cursor_spid
    retvar

    msg emsg "The number of lock(s) is " locks
    return
```

1.6 ERROR AND STATUS INFORMATION

See JAM/DB*i* *Manual – Section 9.2 and Chapter 13*

In Release 5, JAM/DB*i* uses the global variables described in the following sections to supply error and status information in an application. Note that some global variables may not be used in the current release; however, these variables are reserved for use in other engines and for use in future releases of JAM/DB*i* for SYBASE.

1.6.1 Errors

JAM/DB*i* initializes the following global variables for error code information:

`@dmretcode`	Standard JAM/DB*i* status code.
`@dmretmsg`	Standard JAM/DB*i* status message.
`@dmengerrcode`	SYBASE error code.
`@dmengerrmsg`	SYBASE error message.
`@dmengreturn`	Return code an executed stored procedure.

SYBASE returns error codes and messages when it aborts a command. It aborts a command usually because the application used an invalid option or because the user did not have the authority required for an operation. JAM/DB*i* writes SYBASE error codes to the global variable `@dmengerrcode` and writes SYBASE messages to `@dmengerrmsg`.

All SYBASE errors with a severity greater than 10 are JAM/DB*i* errors. Otherwise, they are considered warnings.

The easiest way to test for SYBASE errors is with an installed error or exit handler. For example,

```
dbms ONERROR JPL errors
dbms DECLARE dbi_session CONNECTION FOR ...

proc errors
parms stmt engine flag
   if @dmengerrcode == 0
      msg emsg 'JAM/DBi error: " @dmretmsg
   else
      msg emsg "JAM/DBi error: " @dmretmsg " %N" \
      ":engine error is" @dmengerrcode " " @dmengerrmsg
   return 1
```

If you need additional information about SYBASE errors, please consult your SYBASE documentation.

1.6.2
Warnings

JAM/DB*i* initializes the following global variables for warning information:

`@dmengwarncode`	SYBASE warning code.
`@dmengwarnmsg`	SYBASE warning message.

A warning usually describes some non-fatal change in the SYBASE environment. For example, SYBASE issues a warning when the application changes a connection's default database.

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

```
proc check_status
parms stmt engine flag

if @dmengwarncode
   msg emsg "SYBASE Warning is " @dmengwarnmsg
return
```

1.6.3
Row Information

JAM/DB*i* initializes the following global variables for row information:

`@dmrowcount`	Count of the number of SYBASE rows affected by an operation.
`@dmserial`	Not used in JAM/DB*i* for SYBASE.

SYBASE returns a count of the rows affected by an operation. JAM/DB*i* writes this value to the global variable `@dmrowcount`.

As explained on the manual page for `@dmrowcount`, the value of `@dmrowcount` after a `SELECT` is the number of rows fetched to JAM variables. This number is less than or equal to the total number of rows in the select set. Immediately after an `INSERT`, `UPDATE`, or `DELETE`, `@dmrowcount` is set to the total number of rows affected by the operation. This variable is cleared whenever a `DBMS COMMIT` statement is executed.

The value of `@dmrowcount` may be unexpected after executing a stored procedure. If the stored procedure executes a `SELECT`, `@dmrowcount` equals the number of rows fetched. If, however, the stored procedure does an `INSERT`, `UPDATE`, or `DELETE`, `@dmrowcount` is set to –1. This is documented SYBASE behavior. If you need this information, SYBASE recommends that you test for it within the stored procedure and return it as an output parameter or return code. `@rowcount` is a SYBASE global variable. For example,

```
create proc update_ship_fee @class int, @change float
as
declare @u_count int
update cost set ship_fee = ship_fee * @change
   where class = @class
select @u_count = @rowcount
return @u_count
```

See your SYBASE Command Reference Manual for more information.

1.7 UTILITIES

See JAM/DB*i* *Manual – Chapter 16*

If you start the utilities in interactive mode using the `-i` flag, the utility displays an engine-independent logon screen. JAM/DB*i* uses the following options:

- `User`
- `Password`
- `Server name`
- `Database name`

when declaring a connection to SYBASE for the utilities. Enter the same information you use to declare a connection in `jamdbi`. The other fields on the logon screen may remain empty.

1.7.1 f2tbl

`f2tbl` creates a database table based on a JAM form. It uses each named field on the form to create a column, translating field edits to an appropriate SYBASE column definition. The table below shows the default SYBASE column definitions for each JAM type.

If you do not know how to check a field's JAM type, please see the *Utility Reference Chapter* of the JAM/DB*i* manual.

JAM *Type*	*SYBASE Column Definition*		
	Type	*Length*	*Precision*
DT_CURRENCY	money		
DT_DATETIME	datetime		
DT_YESNO	char	Same as field length	
FT_CHAR	char	Same as field length	
FT_DOUBLE	float		
FT_FLOAT	float		
FT_INT	int		
FT_LONG	int		
FT_PACKED	float		
FT_SHORT	smallint		
FT_UNSIGNED	int		
FT_VARCHAR	varchar	Same as field length	
FT_ZONED	float		

The utility assigns a length for character-type columns. For all other columns, it uses the default length of the datatype.

To change these defaults you must edit the JPL procedure `type` in the distribution JPL module `sybf2t.jpl`, compile it by using `jpl2bin`, and replace the previous version in `sybjpl.lib` by using `formlib -r`.

1.7.2
tbl2f

`tbl2f` creates a JAM form based on an SYBASE table. It creates a field for each column in the table, using the column's datatype to assign the appropriate field characteristics. The

table below lists the following for each SYBASE datatype: the identification number for that datatype from the SYBASE system table `systypes`, the default JAM type and the default field length and precision.

JAM/DB*i* by default preserves the same case when creating field names for a `tbl2f` screen. This is consistent with the default case setting for SYBASE in `dbiinit.c` (see Section 1.1). If you changed the default in `dbiinit.c` to `DM_FORCE_TO_LOWER_CASE` or `DM_FORCE_TO_UPPER_CASE`, you should set the case option of `tbl2f` to match. The case option may be set on the command line or from a pull-down menu in interactive mode. For example, to start `tbl2f` in interactive mode and use upper case for JAM variables, type

```
tbl2f -i -lu
```

Note that there are additional characteristics associated with each JAM type. Those are described in the *Utility Reference Chapter* of the JAM/DB*i* manual.

SYBASE Type	JAM *Field Definition*		
	JAM *Type*	*Length*	*Precision*
smallint 52	FT_SHORT	6	
tinyint 48	FT_SHORT	3	
timestamp, varbinary 37	FT_UNSIGNED		
int 45	FT_UNSIGNED	11	
bit 50	FT_UNSIGNED	11	
int 56	FT_LONG	11	
intn 38	FT_LONG	11	
float 62	FT_FLOAT	25	5
floatn 59, 109	FT_FLOAT	25	5
char 47	FT_CHAR		
money 60	DT_CURRENCY	11	
moneyn 110, 122	DT_CURRENCY	11	
datetime 58, 61	DT_DATETIME	20	

SYBASE Type	JAM *Field Definition*		
	JAM *Type*	*Length*	*Precision*
datetimen 111	DT_DATETIME	20	
varchar 35	FT_VARCHAR	255	
sysname, varchar 39	FT_VARCHAR		

To change these defaults, or to add other datatypes, you must edit the JPL procedure `type` in the distribution JPL module `sybt2f.jpl`, compile it by using `jpl2bin`, and replace the previous version in `sybjpl.lib` by using `formlib -r`.

1.8
STORED PROCEDURES

An application may execute a stored procedure with the command `sql` and the engine's command for execution. For example,

```
sql EXEC procedure
```

executes the named stored procedure. An application may also use a named cursor to execute a stored procedure.

```
dbms DECLARE cursor CURSOR FOR \
   [declare parameter type [declare parameter type...]]\
   EXEC procedure [ parameter [OUT], [parameter [OUT]...] ]
dbms [WITH CURSOR cursor] EXECUTE [USING values]
```

For example, if `emp_grades` is the following stored procedure,

```
create proc emp_grades @gval char(1)
as
select last, first from emp where grade = @gval
```

either of the following,

```
sql EXEC emp_grades :+grade
```

or

```
dbms DECLARE x CURSOR FOR EXEC emp_grades ::g_parm
dbms WITH CURSOR x EXECUTE USING grade
```

executes the stored procedure, selecting the names of all employees with the specified grade. If the current screen (or LDB) contains the fields `last` and `first`, the procedure writes the values to JAM.

Remember, double colons (::) in a `DECLARE CURSOR` statement are for cursor parameters. A value is supplied for the employee grade each time the cursor is executed. If a single colon or colon-plus were used, the employee grade would be supplied when the cursor was declared, not when it was executed. See Section 8.2 in the JAM/DB*i* manual for more information.

If the DBMS supports output parameters, the keyword `OUT` traps the value of an output parameter in a JAM variable. For example, if `summ_by_grade` is the following stored procedure,

```
create proc summ_by_grade
  @cnt int output, @asal money output, @gr char(1)
as
create table empsum (ss char(11), sal money)
insert into empsum select emp.ss, acc.sal from emp, acc
   where emp.ss=acc.ss and emp.grade = @gr
select @cnt = count(*) from empsum
select @asal = avg(sal) from empsum
drop table empsum
```

the application should declare a cursor for the procedure:

```
dbms DECLARE curl CURSOR FOR \
   declare @t1 int declare @t2 money \
   EXEC summ_by_grade @cnt=@t1 OUT, @asal=@t2 OUT, \
   @gr=::grade_parm
dbms WITH CURSOR curl EXECUTE USING gr = grade
```

If `cnt` and `asal` are JAM variables, the procedure returns the number of employees in the specified grade and their average salary. Note that `t1` and `t2` are temporary SYBASE variables, not JAM variables. SYBASE requires that output values be passed as variables, not as constants. The application may use `DBMS ALIAS` to map the values of output parameters to JAM variables. For example,

```
dbms DECLARE curl CURSOR FOR \
   declare @t1 int declare @t2 money \
   EXEC summ_by_grade @cnt=@t1 OUT, @asal=@t2 OUT, \
   @gr=::grade_parm
dbms WITH CURSOR curl ALIAS cnt emp_count, asal sal_avg
dbms WITH CURSOR curl EXECUTE USING gr = grade
```

maps the value of `cnt` to the JAM variable `emp_count` and the value of `asal` to the JAM variable `sal_avg`.

1.8.1

Remote Procedure Calls

In addition to the `EXEC` command, SYBASE supports a remote procedure call ("rpc") for executing a stored procedure. Developers should consider using rpc rather than `EXEC` when either the following occur:

- One or more of the stored procedure's parameters has a datatype that is not `char`. An rpc is more efficient in these cases because it is capable of passing parameters in their native datatypes rather than only as ASCII characters. This reduces the amount of data conversion for the application and the server.
- The stored procedure returns output parameters. An rpc provides a faster and simpler mechanism for accommodating output parameters.

To make an remote procedure call, an application performs the following steps:

1. Must declare an rpc cursor.
2. Must declare the datatype of each parameter that has a non-char datatype.
3. May specify aliases for output parameters or selected columns.
4. Must execute the cursor, supplying in the `USING` clause a JAM variable for each parameter.

The sections below describe these steps in detail. Examples follow.

Declaring the rpc Cursor

JAM/DB*i* uses binding to support rpcs. Therefore, to execute a stored procedure with an rpc, the application must declare an rpc cursor. The syntax is the following:

```
dbms [WITH CONNECTION connection] \
   DECLARE cursor CURSOR FOR \
   RPC procedure [::parameter [OUT] [, ::parameter [OUT]...]]
```

The keyword `RPC` is required. Following the keyword is the name of the procedure and the names of the procedure's parameters. All parameters must begin with a double colon, the JAM/DB*i* syntax for cursor parameters. If a parameter is an output parameter, the keyword `OUT` should follow the parameter name if the application is to receive its value.

Datatyping the rpc Parameters

To pass parameters in their native datatypes, the application must specify a datatype for each non-character parameter. The syntax for DBMS TYPE is the following:

```
dbms [WITH] CURSOR cursor TYPE [parameter] engine_datatype \
    [, [parameter] engine_datatype ...]
```

parameter is a parameter in the DBMS DECLARE CURSOR statement. *engine_datatype* is the datatype of a parameter in the procedure. If parameter names are not given, the types are assigned by position.

JAM/DB*i* uses the information in the DBMS TYPE statement to make the required calls to add parameters to an rpc. Please note that DBMS TYPE has no effect on the data formatting performed for binding.

Redirecting the Value of Output Parameter

By default, when an rpc cursor with an output parameter is executed, JAM/DB*i* searches for a JAM variable with the same name as the output parameter. To write the output value to a JAM variable with another name, use the DBMS ALIAS command.

```
dbms [WITH] CURSOR cursor ALIAS [output_parameter] jamvar \
    [, [output_parameter] jamvar ...]
```

If the procedure selects rows, aliases may be given for the tables' columns. If the procedure returns output parameters and column values, aliases should be given by name rather than by position.

Executing the rpc Cursor

The application executes the stored procedure by executing the rpc cursor. The USING clause must provide a JAM variable for each parameter. The syntax is the following:

```
dbms [WITH] CURSOR cursor EXECUTE \
    USING [parameter =] variable [, [parameter =] variable ...]
```

JAM/DB*i* passes the name of parameter given in the DBMS DECLARE CURSOR statement, the datatype of the parameter given in the DBMS TYPE statement, and the parameter's value which is the value of *variable*.

Parameters and JAM variables may be bound either by name or by position. The two forms should not be mixed, however, in one statement.

Example

If `newsal` is the following stored procedure,

```
create proc newsal
  @ssn char(11), @change float,
  @salary money output, @proposed_sal money output
as
select @salary = (select sal from acc where ssn = @ssn)
select @proposed_sal = @salary * (@change + 1)
```

an rpc would be more efficient than an exec cursor because the procedure has an input parameter with a non-char datatype, and because it returns two output parameters.

The following statement declares an rpc cursor for the stored procedure. Note that the keyword OUT follows each of the output parameters.

```
dbms DECLARE cur2 CURSOR FOR RPC newsal ::ssn, ::change,\
   ::salary OUT, ::proposed_sal OUT
```

Before executing the cursor, the application must specify the SYBASE datatypes for the three non-character datatypes.

```
dbms WITH CURSOR cur2 TYPE \
   change float, salary money, proposed_sal money
```

When executing the cursor, the application must provide a JAM variable for each parameter. JAM/DB*i* passes the name, datatype, and value of the parameters to the procedure. Note that the procedure does not use the input value of the parameters `salary` and `proposed_sal`. JAM/DB*i*'s binding mechanism, however, requires a variable in the `USING` clause for each parameter.

```
dbms WITH CURSOR cur2 EXECUTE \
   USING ssn, change, salary, proposed_sal
```

The procedure passes its output, the two salary values, to the JAM variables `salary` and `proposed_sal`. To put the output values in the fields `sal1` and `sal2`, execute the following:

```
dbms WITH CURSOR cur2 ALIAS salary sal1, \
   proposed_sal sal2
dbms WITH CURSOR cur2 EXECUTE USING ssn=ssn, \
   change=change, salary=currency, proposed_sal=currency
```

Note that the variable names in the `USING` clause do not affect the destination of output values when the cursor is executed. Only a `DBMS ALIAS` statement can remap the output variables to other JAM variables.

Of course, this procedure may also be executed with the standard `EXEC` cursor. It would require the following declaration,

```
dbms DECLARE cur3 CURSOR FOR \
  declare @x money declare @y money \
  EXEC newsal @ssn = ::ssn, @change = ::change, \
  @salary = @x output, @proposed_sal = @y output

dbms WITH CURSOR cur3 EXECUTE USING ssn=ssn, change=change
```

1.8.2

Controlling the Execution of a Stored Procedure

JAM/DB*i* provides a command for controlling the execution of a stored procedure that contains more than one SELECT statement. The command is

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

where ***behavior*** is one of the following

```
STOP_AT_FETCH

EXECUTE_ALL
```

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

The application may execute

```
dbms [WITH CURSOR cursor] CONTINUE
```

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

```
dbms [WITH CURSOR cursor] FLUSH
```

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

```
dbms [WITH CURSOR cursor] NEXT
```

DBMS NEXT automatically flushes any pending SELECT rows.

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

```
dbms [WITH CURSOR cursor] CANCEL
```

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

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

1.8.3

Trapping a Return Code from a Stored Procedure

JAM/DB*i* provides the global variable

```
@dmengreturn
```

to trap the return status code of a stored procedure. This variable is empty unless a stored procedure explicitly sets it. Note that the variable will not be set until the procedure has completed execution. Therefore, an application should evaluate `@dmengreturn` when `@dmretcode = DM_END_OF_PROC`. See Appendix B in the JAM/DB*i* manual for the value of DM_END_OF_PROC.

Executing a new `sql` or `dbms` statement, clears the value of `@dmengreturn`.

If `multiply` is the following stored procedure,

```
create proc multiply @m1 int, @m2 int,
   @guess int output, @result int output
as
select @result = @m1 * @m2
if @result = @guess
   return 1
else
   return 2
```

the application should set up variables for the output parameters.

Either an rpc cursor or an exec cursor may be declared and executed for the procedure,

```
# RPC cursor
dbms DECLARE x CURSOR FOR \
   RPC multiply ::m1, ::m2, ::guess OUT, ::result OUT
dbms WITH CURSOR x TYPE m1 int, m2 int, \
   guess int, result int
dbms WITH CURSOR x ALIAS guess attempt, result answer
dbms WITH CURSOR x EXECUTE USING m1, m2, attempt, answer
# EXEC cursor
dbms DECLARE y CURSOR FOR \
   declare @syb_tmp1 int \
   declare @syb_tmp2 int \
   select @syb_tmp1 = ::user_guess\
   EXEC multiply @m1=::p1, @m2=::p2, \
      @guess= @syb_tmp1 OUT, @result= @syb_tmp2 OUT
dbms WITH CURSOR y ALIAS guess attempt, result answer
dbms WITH CURSOR y EXECUTE \
   USING user_guess = attempt, p1 = m1, p2 = m2
```

After executing the cursor, the application may test the value of `@dmengreturn` and display a message based on the return status code.

```
proc check_ret
# DM_END_OF_PROC is a constant in the LDB.
if @dmretcode == DM_END_OF_PROC
{
   if @dmengreturn == 1
      msg emsg "Good job!"
   else if @dmengreturn == 2
      msg emsg "Better luck next time."
}
else
{
   dbms NEXT
   jpl check_ret
}
return
```

1.9
TRANSACTIONS

On SYBASE, a transaction controls exactly one cursor. Therefore, in a JAM/DB*i* application a transaction controls all statements executed with a single named cursor or the default

cursor. Applications that need transaction control on multiple cursors should use two-phase commit service. The discussion of the JAM/DB*i* commands for two-phase commit is in Section 1.9.2.

The following events commit a transaction on SYBASE:

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

The following events rollback a transaction on SYBASE:

- executing a `DBMS ROLLBACK`.
- closing the transaction's cursor or connection before the transaction is committed

Note that SYBASE will not rollback remote procedure calls (rpcs) or data definition commands that create or drop database objects. See the SYBASE documentation for more information on these restrictions.

1.9.1

Transaction Control on a Single Cursor

Once a connection has been declared, an application may begin a transaction on the default cursor or on any declared cursor.

SYBASE supports the following transaction commands:

- `DBMS [WITH CONNECTION` ***connection***`] BEGIN`
  `DBMS [WITH CURSOR` ***cursor***`] BEGIN`
  Begin a transaction on a default or named cursor.
- `DBMS [WITH CONNECTION` ***connection***`] SAVE` ***savepoint***
  `DBMS [WITH CURSOR` ***cursor***`] SAVE` ***savepoint***
  Create a savepoint in the transaction on a default or named cursor.
- `DBMS [WITH CONNECTION` ***connection***`] COMMIT`
  `DBMS [WITH CURSOR` ***cursor***`] COMMIT`
  Commit the transaction on a default or named cursor.
- `DBMS [WITH CONNECTION` ***connection***`] ROLLBACK [`***savepoint***`]`
  `DBMS [WITH CURSOR` ***cursor***`] ROLLBACK [`***savepoint***`]`
  Rollback to a savepoint or to the beginning of the transaction on a default or named cursor.

A transaction on a default cursor controls all inserts, updates, and deletes executed with the JPL command `sql` or `dm_sql`. The application may set the default connection before beginning the transaction or it may use the `WITH CONNECTION` clause in each statement. A simple transaction on a default cursor may appear as

```
dbms CONNECTION connection
dbms BEGIN
sql statement
sql statement
...
dbms SAVE savepoint
sql statement
dbms ROLLBACK savepoint
dbms COMMIT
```

If a named cursor is declared for multiple statements, it may be useful to execute the cursor in a transaction. This way the application may ensure that SYBASE executes either all of the cursor's statements or none of the cursor's statements. A simple transaction on a named cursor may appear as

```
dbms DECLARE cursor CURSOR FOR statement [statement...]
dbms WITH CURSOR cursor BEGIN
dbms WITH CURSOR cursor EXECUTE [USING parm [parm...]]
...
dbms WITH CURSOR cursor COMMIT
```

If necessary, the cursor may be executed more than once in the transaction. The application should not, however, redeclare a cursor within a transaction.

Examples are shown below with error handlers.

Example 1. A Transaction on the Default Cursor

```
# Call the transaction handler and pass it the name
# of the subroutine containing the transaction commands.
jpl tran_handle new_employee
```

```
proc tran_handle
{
    parms subroutine
    vars jpl_retcode
    retvar jpl_retcode
# Call the subroutine.
        jpl :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 == 0
        {
            msg emsg "Transaction succeeded."
        }
        else
        {
            msg emsg "Aborting transaction."
            dbms ROLLBACK
        }
}


proc new_employee
dbms BEGIN
    sql INSERT INTO emp VALUES \
        (:+ssn, :+last, :+first, \
        :+street, :+city, :+st, :+zip)
    sql INSERT INTO review VALUES \
        (:+ssn, :+startdate, :+startsal, :+grade)
    sql INSERT INTO acc VALUES (:+ssn, :+startsal, :+exmp)
dbms COMMIT
return 0
```

The procedure `tran_handle` is a generic handler for the application's transactions. It is like the one described in the *Developer's Guide*. The procedure `new_employee` contains the transaction statements. This method reduces the amount of error checking code.

The application executes the transaction by executing

```
jpl tran_handle new_employee
```

The procedure `tran_handle` receives the argument "new_employee" and writes it to the variable `subroutine`. It defines and declares a JPL variable to receive a JPL return code. After performing colon processing `:subroutine` is replaced with its value,

new_employee, and JPL calls the procedure. The procedure new_employee begins the transaction, performs three inserts, and commits the transaction.

If new_employee executes without any errors, it returns 0 to the variable jpl_retcode in the calling procedure tran_handle. JPL then evaluates the if statement, displays a success message, and exits.

If however an error occurs while executing new_employee, JAM/DB*i* calls the application's error handler. The error handler should display any error messages and return the abort code, 1.

For example, assume the first INSERT in new_employee executes successfully but the second INSERT fails because it violates the rule grade_range. In this case, JAM/DB*i* calls the error handler to display an error message. When the error handler returns the abort code 1, JAM aborts the procedure new_employee (therefore, the third INSERT is not attempted). JAM returns 1 to jpl_retcode in the calling procedure tran_handle. JPL evaluates the if statement, displays a message and executes a rollback. The rollback undoes the insert to the table emp.

1.9.2
Transaction Control on Multiple Cursors

SYBASE provides two-phase commit service for distributed transactions. In a two-phase commit, one main transaction controls two or more subtransactions on one or more servers. A subtransaction is a transaction on single cursor, like those described in the section above.

With two-phase commit service using Microsoft SQL Server, the commit server and the target server must be different servers.

The main transaction must be declared with the command

```
dbms [WITH CONNECTION connection] \
    DECLARE transaction TRANSACTION FOR \
    APPLICATION application SITES sites
```

- ***connection***: if no connection is given, the default connection is used; the connection data structure stores a user login name, a server name, and an interface file name. Since SYBASE requires that a particular server be responsible for coordinating a two-phase commit, the connection declaration must include a server name.
- ***transaction***: the name of the transaction; SYBASE does not permit periods (.) or colons (:) in a transaction name. Since "transaction" and "tran" are keywords for both JAM/DB*i* and SYBASE, do not use these words for this argument.

- ***application***: the name of the application; it may be any character string that is not a keyword.
- ***sites***: the number of cursors (i.e., subtransactions) participating in the two-phase commit. This value is used by the SYBASE commit and recovery systems and must be set appropriately.

Once the two-phase commit transaction is declared, its name is used to begin and to commit or rollback the transaction. The syntax is

```
dbms BEGIN transaction
dbms COMMIT transaction
dbms ROLLBACK transaction
```

As with cursors and connections, JAM/DB*i* uses a data structure to manage a two-phase commit transaction. This structure should be closed when the transaction is completed. When the structure is closed, JAM/DB*i* calls the support routine to close the connection with the SYBASE commit service. The command is the following:

```
dbms CLOSE TRANSACTION transaction
```

Operations on a single cursor are subtransactions. To control a subtransaction in a two-phase commit transaction, the following commands may be used:

```
dbms [WITH CURSOR cursor] BEGIN
dbms [WITH CURSOR cursor] SAVE savepoint
dbms [WITH CURSOR cursor] PREPARE_COMMIT
dbms [WITH CURSOR cursor] COMMIT
dbms [WITH CURSOR cursor] ROLLBACK [savepoint]
```

The command `DBMS PREPARE_COMMIT` is an additional command required by the two-phase commit service. Executing it signals that the subtransaction has been performed and that the server is ready is to commit the update. Once the application has "prepared" all the subtransactions, it issues a `COMMIT` to the main transaction and each subtransaction.

The sequence of events in a SYBASE two-phase commit transaction is the following:

1. Declare any necessary connections and cursors.
2. Declare the main transaction.

   ```
   dbms DECLARE tname TRANSACTION FOR SITES sites \
     APPLICATION application
   ```

3. Begin the main transaction.

   ```
   dbms BEGIN tname
   ```

4. For each subtransaction cursor, begin the subtransaction and execute the desired operations. When all subtransactions are complete, execute a `PREPARE_COMMIT` for each. In the pseudo code below there are three subtransactions (using `cursor1`, the default cursor, and `cursor2`):

```
dbms WITH CURSOR cursor1 BEGIN
dbms WITH CURSOR cursor1 EXECUTE USING parm

dbms BEGIN
sql statement
sql statement
dbms SAVE savepoint
sql statement
dbms ROLLBACK savepoint

dbms WITH CURSOR cursor2 BEGIN
dbms WITH CURSOR cursor2 EXECUTE USING parm

dbms WITH CURSOR cursor1 PREPARE_COMMIT
dbms PREPARE_COMMIT
dbms WITH CURSOR cursor2 PREPARE_COMMIT
```

5. Commit the main transaction.

```
dbms COMMIT tname
```

6. Commit each subtransaction indicating a named or default cursor.

```
dbms WITH CURSOR cursor1 COMMIT
dbms COMMIT
dbms WITH CURSOR cursor2 COMMIT
```

7. Close the transaction.

```
dbms CLOSE TRANSACTION tname
```

It is strongly recommended that the application use an error handler while the transaction is executing. If an error occurs while executing a command in the subtransaction (i.e., executing a `sql` statement or a named cursor) the application should not continue executing the transaction.

An example with an error handler follows.

```
############################################################
# Declare connections and specify servers.
dbms DECLARE c1 CONNECTION \
   FOR USER :uid PASSWORD :pwd SERVER maple \
   INTERFACES '/usr/sybase/interfaces.ny'
dbms DECLARE c2 CONNECTION \
   FOR USER :uid PASSWORD :pwd SERVER juniper
```

```
# Declare cursors.
# Use :: to insert a value when the cursor is executed,
# not when the cursor is declared.
dbms WITH CONNECTION c1 DECLARE x CURSOR FOR INSERT \
   emp (ss, last, first, street, city, st, zip, grade) \
   VALUES (::ss, ::last, ::first, ::street, ::city, \
   ::st, ::zip, ::grade)
dbms WITH CONNECTION c2 DECLARE y CURSOR FOR INSERT \
   acc (ss, sal, exmp) VALUES (::ss, ::sal, ::exmp)

############################################################
proc 2phase
vars retval
call sm_s_val
if retval
{
   msg reset "Invalid entry."
   return
}
dbms WITH CONNECTION c1 DECLARE new_emp TRANSACTION \
   FOR APPLICATION personnel SITES 2
dbms ONERROR JPL tran_error
jpl do_tran
if !(retval)
   msg emsg "Transaction succeeded."
else
{
   dbms ROLLBACK newemp
   if retval >= 100
      dbms WITH CURSOR x ROLLBACK
   if retval >= 200
      dbms WITH CURSOR y ROLLBACK
}
dbms ONERROR CALL generic_errors
dbms CLOSE TRANSACTION new_emp
return

proc do_tran
# Begin new_emp and set the flag tran_level (LDB var)
dbms BEGIN new_emp

   dbms WITH CURSOR x BEGIN
   cat tran_level "1"
   dbms WITH CURSOR x EXECUTE USING \
      (ss, last, first, street, city, st, zip, grade)
```

```
    dbms WITH CURSOR y BEGIN
    cat tran_level "2"
    dbms WITH CURSOR y EXECUTE USING \
       (ss, startsal, exemptions)

    dbms WITH CURSOR x PREPARE_COMMIT
    dbms WITH CURSOR y PREPARE_COMMIT

# Execute commits.
dbms COMMIT new_emp
    dbms WITH CURSOR x COMMIT
    dbms WITH CURSOR y COMMIT

msg emsg "Insert completed."
cat tran_level ""
return

############################################################
proc tran_error
vars fail_area [2](20), tran_err(3)
cat fail_area[1] "address"
cat fail_area[2] "accounting data"

if tran_level != ""
{
    # Display an error message describing the failure.
    msg emsg "%WTransaction failed. Unable to insert \
          :fail_area[tran_level] because of " @dmengerrmsg
    math tranerr = tran_level * 100
    cat tran_level ""
    return :tranerr
}
msg emsg @dmengerrmsg
return 1
```

1.10
SYBASE-SPECIFIC COMMANDS

See JAM/DB*i* *Manual – Chapter 11*

JAM/DB*i* for SYBASE provides additional commands for SYBASE-specific features. If you are using multiple engines or are porting an application to or from another engine, please note that these commands may work differently or may not be supported on some engines.

1.10.1
Using Browse Mode

BROWSE	execute a SELECT for browsing
UPDATE	update a table while browsing

1.10.2
Using Stored Procedures

CANCEL	abort execution of a stored procedure
DECLARE CURSOR FOR RPC	declare a cursor to execute a stored procedure using a remote procedure call
FLUSH	abort execution of a stored procedure
NEXT	execute the next statement in a stored procedure
SET	set execution behavior for a procedure (execute all, stop at fetch, etc.)
TYPE	set data types for parameters of a stored procedure executed with an rpc cursor

1.10.3
Using Transactions

JAM/DB*i* supports the following commands when using transactions. See the reference pages for more information on each command.

BEGIN	begin a transaction
CLOSE_ALL_TRANSACTIONS	close all transactions declared for two-phase commit
CLOSE TRANSACTION	close a named transaction
COMMIT	commit a transaction
DECLARE TRANSACTION	declare a transaction for two-phase commit
PREPARE_COMMIT	prepare to commit a transaction

ROLLBACK	rollback a transaction
SAVE	save a two-phase commit

BEGIN

start a transaction

SYNOPSIS

```
dbms [WITH CONNECTION connection] BEGIN
dbms [WITH CURSOR cursor] BEGIN

dbms  BEGIN two_phase_commit
```

DESCRIPTION

This command sets the starting point of a transaction. It is available in two contexts. It can start a transaction on a single cursor or it can start a distributed transaction which may involve multiple cursors on different servers.

A transaction is a logical unit of work on a database contained within `DBMS BEGIN` and `DBMS COMMIT` statements. `DBMS BEGIN` defines the start of a transaction. Once a transaction is begun, changes to the database are not committed until a `DBMS COMMIT` is executed. Changes are undone by executing `DBMS ROLLBACK`.

If a `WITH CURSOR` clause is used in a `DBMS BEGIN` statement, JAM/DB*i* begins a transaction on the named cursor. If a `WITH CONNECTION` clause is used, JAM/DB*i* begins a transaction on the default cursor of the named connection. If no `WITH` clause is used, JAM/DB*i* begins a transaction on the default cursor of the default connection.

To begin a distributed transaction (two-phase transaction), first declare a named transaction with `DBMS DECLARE TRANSACTION`. Since this statement supports a `WITH CONNECTION` clause, JAM/DB*i* associates the transaction name with a particular connection; the connection's server is the coordinating server for the distributed transaction. When the application executes `DBMS BEGIN` ***two_phase_commit*** where ***two_phase_commit*** is the name of the declared transaction, JAM/DB*i* starts the transaction on the coordinating server.

Be sure to terminate the transaction with a `DBMS ROLLBACK` or `DBMS COMMIT` before logging off. Note that JAM/DB*i* will not close a connection with a pending two-phase commit transaction.

SEE ALSO

Section 1.9 – Transactions.

Documentation provided by the database vendor.

RELATED COMMANDS

dbms COMMIT

dbms ROLLBACK

dbms SAVE

EXAMPLE

Refer to the examples in *Section 1.9 – Transactions.*

BROWSE

retrieve SELECT results one row at a time

SYNOPSIS

dbms BROWSE ***SELECTstmt***

DESCRIPTION

This command allows an application to execute a SELECT in "browse" mode. This means that SYBASE will return the SELECT rows one at a time to the JAM/DB*i* application; SYBASE will not set any shared locks for the SELECT. The application may use the companion command DBMS UPDATE to update the current row. SYBASE will verify that the row has not been changed before it issues the UPDATE.

To use browse mode, the table being updated must have a timestamp column and a unique index. A row's timestamp indicates the last time the row was updated. If the timestamp has not changed since DBMS BROWSE was executed, the application may UPDATE the row. If the timestamp has changed, then some other user or application has updated the row after DBMS BROWSE was executed. The update is aborted and an error is returned.

Browse mode requires a connection with two default cursors. The application must open the browse mode connection by setting the CURSORS option to 2. JAM/DB*i* uses one default cursor to select the rows and the other default cursor to update the rows.

It is the programmer's responsibility to determine whether a table is browsable. It the table is not browsable, JAM/DB*i* returns the DM_BAD_ARGS error. If a table is browsable, JAM/DB*i* returns the first row in the select set when DBMS BROWSE is executed. Note that only row is returned at a time.

To view the next row, the application must execute DBMS CONTINUE.

RELATED COMMANDS

dbms CONTINUE

dbms FLUSH

dbms UPDATE

EXAMPLE

```
# Browse mode requires a connection declared with 2
# cursors.
dbms DECLARE browse_con CONNECTION FOR \
   USER :user PASSWORD :pass SERVER :server CURSORS 2
```

```
proc start_browse_mode
   dbms CONNECTION browse_con
   dbms BROWSE SELECT ss, last, first, sal FROM employee
   return

proc update_browse_row
# Allow the user to update the employee salary. DBi builds
# the WHERE clause to identify this row.
   dbms UPDATE employee SET sal = :+sal
   return

proc next_browse_row
# Fetch the next row.
   dbms CONTINUE
   return
```

CANCEL

cancel the execution of a stored procedure

SYNOPSIS

```
dbms [WITH CURSOR cursor] CANCEL
```

DESCRIPTION

This command cancels any outstanding work on the named cursor. In particular, this command may be used to cancel a pending stored procedure. When the statement is executed, the following operations are performed:

- any rows to be fetched are flushed
- any remaining unexecuted statements are ignored
- the procedure's return status code is returned

If the `WITH CURSOR` clause is not used, JAM/DB*i* executes the command on the default cursor.

SEE ALSO

- *Section 1.8 – Stored Procedures.*

RELATED COMMANDS

`dbms FLUSH`

CLOSE_ALL_TRANSACTIONS

close all transactions declared for two-phase commit

SYNOPSIS

```
dbms CLOSE_ALL_TRANSACTIONS
```

DESCRIPTION

This command attempts to close all transactions declared for two-phase commit with `DBMS DECLARE TRANSACTION`. If the transaction has not been terminated by a `COMMIT` or `ROLLBACK`, JAM/DB*i* will return the error `DM_TRAN_PENDING`.

If an application terminates with a pending two-phase commit transaction, SYBASE will mark the transaction's process as "infected." You will need the system administrator to delete the infected process. To help prevent this, JAM/DB*i* will not close a connection unless all two-phase commit transactions have been closed. Furthermore, JAM/DB*i* will not close a two-phase commit transaction unless the application explicitly terminated the transaction with a `DBMS COMMIT` ***two_phase_commit*** or `DBMS ROLLBACK` ***two_phase_commit***.

Since this command verifies that all two-phase commit transactions were terminated, you may wish to call this command before logging off.

SEE ALSO

Section 1.9 – Transactions.

RELATED COMMANDS

`dbms BEGIN`

`dbms CLOSE TRANSACTION`

`dbms COMMIT`

`dbms DECLARE TRANSACTION`

`dbms ROLLBACK`

EXAMPLE

```
proc cleanup
   dbms ONERROR JPL cleanup_failure
   dbms CLOSE_ALL_TRANSACTIONS
   dbms CLOSE_ALL_CONNECTIONS
   return
```

```
# APP1 = ^jpl two_phase_cleanup
proc cleanup_failure
   parms stmt engine flag
   if @dmretcode == DM_TRAN_PENDING
   {
      call jm_keys APP1
   }
   return 0


proc two_phase_cleanup
   dbms ROLLBACK ...
   dbms CLOSE TRANSACTION ...
   return
```

CLOSE TRANSACTION

close a declared transaction structure

SYNOPSIS

dbms CLOSE TRANSACTION ***transaction***

DESCRIPTION

This command closes the main transaction which was previously defined using DBMS DECLARE TRANSACTION. A main transaction controls the execution of a two–phase commit process. This command signals the completion of the main transaction and closes the SYBASE structures associated with the transaction.

An error code is returned if a transaction was pending. An application cannot close a connection with an open transaction.

SEE ALSO

Section 1.9 – Transactions.

RELATED COMMANDS

dbms BEGIN

dbms COMMIT

dbms DECLARE TRANSACTION

dbms PREPARE_COMMIT

dbms ROLLBACK

dbms SAVE

COMMIT

commit a transaction

SYNOPSIS

```
dbms [WITH CONNECTION connection] COMMIT
dbms [WITH CURSOR cursor] COMMIT
dbms COMMIT two_phase_commit
```

DESCRIPTION

Use this command to commit a pending transaction. Committing a transaction saves all the work since the last `COMMIT`. Changes made by the transaction become visible to other users. If the transaction is terminated by `DBMS ROLLBACK`, the updates are not committed, and the database is restored to its state prior to the start of the transaction.

This command is available in two contexts. It can commit a transaction on a single cursor or it can commit a two–phase commit transaction. If a `WITH CURSOR` clause is used in a `DBMS COMMIT` statement, JAM/DB*i* commits the transaction on the named cursor. If a `WITH CONNECTION` clause is used, JAM/DB*i* commits the transaction on the default cursor of the named connection. If no `WITH` clause or no distributed transaction name is used, JAM/DB*i* commits the transaction on the default cursor of the default connection.

If a distributed transaction name is used, JAM/DB*i* issues the commit to the coordinating server. If this is successful, the application should issue a `DBMS COMMIT` for each subtransactions. A `WITH CURSOR` or `WITH CONNECTION` clause is required for a subtransaction on a named cursor or a subtransaction on the default cursor of a non-default connection.

SEE ALSO

Section 1.9 – Transactions.

RELATED COMMANDS

```
dbms BEGIN
dbms CLOSE TRANSACTION
dbms DECLARE TRANSACTION
dbms PREPARE_COMMIT
dbms ROLLBACK
dbms SAVE
```

EXAMPLE

Refer to the example in *Section 1.9 – Transactions.*

DECLARE CURSOR FOR RPC

declare a named cursor for a remote procedure

SYNOPSIS

```
dbms [WITH CONNECTION connection] DECLARE cursor CURSOR \
   FOR RPC procedure [::parameter [OUT] [datatype] \
   [, ::parameter [OUT] [datatype] ...]]
```

DESCRIPTION

Use this command to create or redeclare a named cursor to execute a remote procedure call (rpc). Since JAM/DB*i* uses its binding mechanism to support rpc's, the default cursor cannot execute an rpc.

The keyword RPC is required. Following the keyword is the name of the procedure and the names of the procedure's parameters. All parameters must begin with a double colon, which is the JAM/DB*i* syntax for cursor parameters. If a parameter is an output parameter, the keyword OUT should follow the parameter name if the application is to receive its value. A parameter's datatype may be given in the DBMS DECLARE CURSOR statement, or in a DBMS TYPE statement.

The application executes an rpc cursor as it executes any named cursor, with DBMS EXECUTE.

SEE ALSO

Section 1.8 – Stored Procedures.

@dmengreturn

RELATED COMMANDS

dbms CLOSE CURSOR

dbms WITH CURSOR *cursor* EXECUTE

dbms TYPE

WITH CURSOR

EXAMPLE

Refer to the example in *Section 1.8 – Stored Procedures.*

DECLARE TRANSACTION

declare a named transaction for two phase commit

SYNOPSIS

```
dbms [WITH CONNECTION connection] \
   DECLARE transaction TRANSACTION FOR \
   SITES sites APPLICATION application
```

DESCRIPTION

This command declares a two-phase commit transaction structure.

The `WITH CONNECTION` clause identifies the server which will coordinate the distributed transaction. If the clause is not used, the server of the default connection is used. Be sure to name the server when declaring the connection.

transaction is the name of the two-phase commit transaction. Do not use the keywords "tran" or "transaction" for this argument. The application will use this name to begin, to commit or rollback, and to close the transaction.

sites is the number of subtransactions involved in the distributed transaction. Each cursor where a `BEGIN` is issued is a subtransaction. This number is critical to recovery if the transaction fails.

application is an optional argument which identifies the name of the transaction.

The application must use ***transaction*** to begin and commit or rollback the two-phase commit.

After declaring the transaction, begin the transaction using `DBMS BEGIN`. When the transaction is complete, close the transaction using either `CLOSE TRANSACTION` or `CLOSE_ALL_TRANSACTIONS`. An application must close all declared transactions before closing their connections.

SEE ALSO

Section 1.9 – Transactions.

RELATED COMMANDS

`dbms CLOSE TRANSACTION` ***transaction***

EXAMPLE

Refer to the examples in *Section 1.9 – Transactions.*

FLUSH

flush any selected rows not fetched to JAM variables

SYNOPSIS

```
dbms [WITH CURSOR cursor] FLUSH
```

DESCRIPTION

Use this command to throw away any unread rows in the SELECT set of the default or named cursor.

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

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

RELATED COMMANDS

dbms DECLARE CURSOR

dbms CANCEL

dbms CONTINUE

dbms NEXT

EXAMPLE

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

NEXT

execute the next statement in a stored procedure

SYNOPSIS

dbms [WITH CURSOR *cursor*] NEXT

DESCRIPTION

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

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

SEE ALSO

Section 1.8 – Stored Procedures.

RELATED COMMANDS

dbms DECLARE CURSOR

dbms CANCEL

dbms CONTINUE

dbms FLUSH

dbms SET [EXECUTE_ALL | SINGLE_STEP | STOP_AT_FETCH]

EXAMPLE

Refer to the example in *Section 1.8 – Stored Procedures.*

PREPARE_COMMIT

prepare a two phase commit

SYNOPSIS

```
dbms [WITH CURSOR cursor] PREPARE_COMMIT
```

DESCRIPTION

Use of this command is required during the two–phase commit service. It needs to be executed for each subtransaction when the subtransaction has been performed. Execution of this command signals the application that the server is ready to commit the update. Once the application has "prepared" all the subtransactions, it needs to issue a `DBMS COMMIT` to the main transaction and to each subtransaction.

If the `WITH CURSOR` clause is not used, JAM/DB*i* issues the command on the default cursor.

SEE ALSO

Section 1.9 – Transactions

RELATED COMMANDS

`dbms BEGIN`

`dbms CLOSE TRANSACTION`

`dbms COMMIT`

`dbms DECLARE TRANSACTION`

`dbms ROLLBACK`

`dbms SAVE`

EXAMPLE

Refer to the example in *Section 1.9 – Transactions.*

ROLLBACK

rollback a transaction

SYNOPSIS

```
dbms [WITH CONNECTION connection] ROLLBACK savepoint
dbms [WITH CURSOR cursor] ROLLBACK savepoint
dbms ROLLBACK two_phase_commit
```

DESCRIPTION

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

This command is available in two contexts. It can rollback a transaction on a single cursor, or it can rollback a two-phase rollback transaction. If a `WITH CURSOR` clause is used in a `DBMS ROLLBACK` statement, JAM/DB*i* rollbacks the transaction on the named cursor. If a `WITH CONNECTION` clause is used, JAM/DB*i* rollbacks the transaction on the default cursor of the named connection. If no `WITH` clause or no distributed transaction name is used, JAM/DB*i* rollbacks the transaction on the default cursor of the default connection.

If a distributed transaction name is used, JAM/DB*i* issues the rollback to the coordinating server. The application should also issue a `DBMS ROLLBACK` for each subtransaction. A `WITH CURSOR` or `WITH CONNECTION` clause is required for a subtransaction on a named cursor or a subtransaction on the default cursor of a non-default connection.

SEE ALSO

Section 1.9 – Transactions

RELATED COMMANDS

```
dbms BEGIN
dbms COMMIT
dbms DECLARE TRANSACTION
dbms PREPARE_COMMIT
dbms ROLLBACK
dbms SAVE
```

EXAMPLE

Refer to the example in *Section 1.9 – Transactions.*

SAVE

set a savepoint or checkpoint within a transaction

SYNOPSIS

```
dbms [WITH CONNECTION connection] SAVE savepoint
dbms [WITH CURSOR cursor] SAVE savepoint
```

DESCRIPTION

This command creates a savepoint in the transaction. A savepoint is a pointer set by the programmer within a transaction. When a savepoint is set, the procedures following the savepoint can be cancelled using `DBMS ROLLBACK` ***savepoint***.

When the transaction is rolled back to a savepoint, the transaction must then be completed or completely rolled back to the beginning.

SEE ALSO

Section 1.9 – Transactions

RELATED COMMANDS

`dbms BEGIN`

`dbms COMMIT`

`dbms DECLARE TRANSACTION`

`dbms PREPARE_COMMIT`

`dbms ROLLBACK`

`dbms SAVE`

EXAMPLE

Refer to the example in *Section 1.9 – Transactions.*

SET

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

SYNOPSIS

```
dbms [WITH CURSOR cursor] SET \
   [EXECUTE_ALL | SINGLE_STEP | STOP_AT_FETCH]
```

DESCRIPTION

This command controls the execution of a stored procedure or a cursor with multiple statements. Its options are

EXECUTE_ALL	Specifies that the DBMS return control to JAM/DB*i* only when all statements have been executed or when an error occurs. If a SELECT is executed, only the first pageful of rows is returned to JAM variables. This option may be set for a multi-statement or a stored procedure cursor.
SINGLE_STEP	Specifies that the DBMS return control to the JAM Executive after executing each statement belonging to the multi-statement cursor. After each SELECT, the user may press a function key to execute a DBMS CONTINUE and scroll the SELECT set. To resume executing the cursor's statements, the application must execute DBMS NEXT. This option may be set for a multi-statement cursor. If this option is used with a stored procedure cursor, JAM/DB*i* uses the default setting STOP_AT_FETCH.
STOP_AT_FETCH	Specifies that the DBMS return control to the JAM Executive after executing a SELECT that fetches rows. (Note that control is not returned for a SELECT that assigns a value to a local SYBASE parameter.) The application may use DBMS CONTINUE to scroll through the SELECT set. To resume executing the cursor's statements or procedure, the application must execute DBMS NEXT. This option may be set for a multi-statement or a stored procedure cursor.

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

SEE ALSO

Section 1.8 – Stored Procedures

RELATED COMMANDS

dbms CANCEL

dbms CONTINUE

dbms DECLARE CURSOR

dbms DECLARE CURSOR FOR EXEC

dbms DECLARE CURSOR FOR RPC

dbms FLUSH

dbms NEXT

EXAMPLE

```
vars DM_NO_MORE_ROWS(5) DM_END_OF_PROC(5)
cat DM_NO_MORE_ROWS "53256"
cat DM_END_OF_PROC "53270"

dbms DECLARE x CURSOR FOR \
   SELECT company, street, city, st, zip \
      FROM client_list WHERE co_id = ::company_id \
   INSERT INTO contacts VALUES \
      (::newfirst, ::newlast, ::newloc, ::newphone) \
   SELECT first, last, location, phone FROM contacts \
      WHERE co_id = ::company_id
msg d_msg "%KPF1 START  %KPF2 SCROLL SELECT\
  %KPF3 EXECUTE NEXT STEP"

proc f1
dbms WITH CURSOR x SET SINGLE_STEP
dbms WITH CURSOR x EXECUTE USING company_id, newfirst, \
   newlast, newloc, newphone, company_id
dbms WITH CURSOR x SET
return

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

```
return

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

SET_BUFFER

set up a buffer for engine-supported scrolling

SYNOPSIS

dbms [WITH CURSOR ***cursor***] SET_BUFFER [***number_of_rows***]

DESCRIPTION

SYBASE supports non-sequential scrolling if the application has set up a buffer for result rows. If an application does not need DBMS CONTINUE_UP or is using a continuation file (DBMS STORE FILE), this command is not needed.

number_of_rows is the number of rows SYBASE will buffer. To be useful, ***number_of_rows*** should be greater than the number of occurrences in the JAM destination fields.

When this command is used with a SELECT cursor, SYBASE saves the specified number of result rows of the SELECT in memory. When the application executes DBMS CONTINUE_BOTTOM, DBMS CONTINUE_TOP, or DBMS CONTINUE_UP commands, the result rows in memory are returned.

The buffer is maintained for the life of the cursor, or until the buffer is released with the command,

dbms [WITH CURSOR ***cursor***] SET_BUFFER

Executing the command without supplying the ***number_of_rows*** argument turns off the feature for the named or default cursor and frees the buffer. Note that redeclaring the cursor does not free the buffer. Closing the cursor does release the buffer.

Because the use of this command is expensive (approximately 2K of memory per row), it should be used only if the application needs non-sequential scrolling but cannot use scrolling arrays or a continuation file. The application should turn off DBMS SET_BUFFER when finished with the SELECT set.

SEE ALSO

dbms STORE [FILE [***filename***]]

RELATED COMMANDS

dbms CONTINUE_BOTTOM

dbms CONTINUE_TOP

dbms CONTINUE_UP

EXAMPLE

```
dbms DECLARE emp_cursor CURSOR FOR SELECT * FROM emp
dbms WITH CURSOR emp_cursor SET_BUFFER 500

proc scroll_up
dbms WITH CURSOR emp_cursor CONTINUE_UP
return

proc scroll_down
dbms WITH CURSOR emp_cursor CONTINUE_DOWN
return
```

TRANSACTION

set a default declared two-phase commit transaction

SYNOPSIS

dbms TRANSACTION ***variable***

DESCRIPTION

If an application has declared more than one two-phase commit transaction, it may use this command to set the default two-phase commit transaction for a subtransaction.

RELATED COMMANDS

dbms BEGIN

dbms COMMIT

dbms DECLARE TRANSACTION

dbms PREPARE_COMMIT

dbms ROLLBACK

dbms SAVE

TYPE

declare parameter datatypes for an rpc cursor

SYNOPSIS

```
dbms WITH CURSOR cursor TYPE parameter datatype \
    [, parameter datatype ...]
```

DESCRIPTION

If an application has declared a cursor for a remote procedure call ("rpc") but has not declared the datatypes of the procedure's parameters, it should use the DBMS TYPE command.

parameter is the name of a parameter in the stored procedure and in the DBMS DECLARE CURSOR statement. ***datatype*** is the datatype of the parameter in the stored procedure. **JAM/DB*i*** uses the information supplied with this command to execute the remote procedure call. Please note that these datatypes have no effect on any data formatting performed by colon-plus processing or binding.

Executing this command with no arguments deletes all type information for the named cursor.

SEE ALSO

Section 1.8 – Stored Procedures

RELATED COMMANDS

```
dbms DECLARE cursor CURSOR FOR RPC procedure \
  [::parameter [OUT] datatype [, ::parameter [OUT] datatype ...]
```

```
dbms DECLARE cursor CURSOR FOR RPC procedure \
  [::parameter [OUT] [, ::parameter [OUT] ...]
```

EXAMPLE

```
############################################################
#procedure newsal:
#create proc newsal @ss char(11), @change float,
# @salary money output, @proposed_sal money output
# as
#  select @salary = (select sal from acc where ss = @ss)
#  select @proposed_sal = @salary * (@change + 1)
############################################################
```

```
dbms DECLARE sal_cursor CURSOR FOR \
   RPC newsal ::ss, ::change, ::salary OUT, \
    ::proposed_sal OUT

dbms WITH CURSOR sal_cursor TYPE \
   change float, salary money, proposed_sal money

dbms WITH CURSOR sal_cursor EXECUTE \
   USING ss, change, salary, proposed_sal
```

UPDATE

update a table while browsing

SYNOPSIS

dbms UPDATE ***table*** SET ***column*** = ***value*** [, ***column*** = ***value*** ...]

DESCRIPTION

Browse mode permits an application to browse through a SELECT set, updating a row at a time. Browse mode is useful for an application that wants to ensure that a row has not been changed in the interval between the fetch and the update of the row.

When DBMS BROWSE is executed, it fetches the rows in the SELECT set one at a time. The application should provide two other procedures to execute DBMS CONTINUE and DBMS UPDATE.

Please note that the DBMS UPDATE statement has no WHERE clause. JAM/DB*i* calls a SYBASE routine to build a where clause using the unique index of the current row and the value of its timestamp column when the row was fetched. If the timestamp value has not been changed, the row is updated. However, if the timestamp value has changed, then another user has modified the row since the application executed DBMS BROWSE; in this case SYBASE will not perform the update.

RELATED COMMANDS

dbms BROWSE

dbms CANCEL

dbms CONTINUE

dbms FLUSH

EXAMPLE

See manual page for DBMS BROWSE.

USE

open an existing database

SYNOPSIS

```
dbms [WITH CONNECTION connection] USE database
```

DESCRIPTION

This command changes a connection's default database. ***database*** must be an existing database, and the user must have the appropriate permissions to use the database or else **JAM/DB*i*** returns an error.

RELATED COMMANDS

```
dbms DECLARE connection CONNECTION FOR [USER user [PASSWORD
password]] [SERVER server] [DATABASE database] [CURSORS [1|2]]
[INTERFACES filename] [TIMEOUT seconds]
```

EXAMPLE

```
dbms DECLARE c1 CONNECTION FOR \
   USER :uname PASSWORD :pword SERVER :server \
   DATABASE master
sql SELECT * FROM emp
dbms WITH CONNECTION c1 USE projects
sql SELECT * FROM newjobs
```

1.11
COMMAND DIRECTORY FOR SYBASE

This section contains a directory for all the commands available in JAM/DB*i* for SYBASE. The following table lists the command, a short description of the command, and the location of the reference page for that command. If the location is described as SYBASE Notes, that information is enclosed in this document.

Command	*Description*	*Documentation*
ALIAS	name a JAM variable as the destination of a selected column or aggregate function	JAM/DB*i* Manual
BEGIN	begin a transaction	SYBASE Notes
BINARY	create a JAM/DB*i* variable for fetching binary values	JAM/DB*i* Manual
BROWSE	execute a SELECT for browsing	SYBASE Notes
CANCEL	abort execution of a stored procedure	SYBASE Notes
CATQUERY	redirect SELECT results to a file or a JAM variable	JAM/DB*i* Manual
CLOSE_ALL_CONNECTIONS	close all connections on all engines	JAM/DB*i* Manual
CLOSE_ALL_TRANSACTIONS	close all transactions	SYBASE Notes
CLOSE CONNECTION	close a named connection	JAM/DB*i* Manual
CLOSE CURSOR	close a cursor	JAM/DB*i* Manual
CLOSE TRANSACTION	close a named transaction	SYBASE Notes
COMMIT	commit a transaction	SYBASE Notes
CONNECTION	set a default connection and engine for the application	JAM/DB*i* Manual

Command	*Description*	*Documentation*
CONTINUE	fetch the next screenful of rows from a SELECT set	JAM/DB*i* Manual
CONTINUE_BOTTOM	fetch the last screenful of rows from a SELECT set	JAM/DB*i* Manual
CONTINUE_DOWN	fetch the next screenful of rows from a SELECT set	JAM/DB*i* Manual
CONTINUE_TOP	fetch the first screenful of rows from a SELECT set	JAM/DB*i* Manual
CONTINUE_UP	fetch the previous screenful of rows from a SELECT set	JAM/DB*i* Manual
DECLARE CONNECTION	declare a named connection to an engine	JAM/DB*i* Manual
DECLARE CURSOR	declare a named cursor	JAM/DB*i* Manual
DECLARE CURSOR FOR RPC	declare a cursor to execute a stored procedure using a remote procedure call	SYBASE Notes
DECLARE TRANSACTION	declare a transaction for two-phase commit	SYBASE Notes
ENGINE	set the default engine for the application	JAM/DB*i* Manual
EXECUTE	execute a named cursor	JAM/DB*i* Manual
FLUSH	abort execution of a stored procedure	SYBASE Notes
FORMAT	format the results of a CATQUERY	JAM/DB*i* Manual
NEXT	execute the next statement in a stored procedure	SYBASE Notes

Command	*Description*	*Documentation*
OCCUR	set the number of rows for JAM/DB*i* to fetch to an array and choose an occurrence where JAM/DB*i* should begin writing result rows	JAM/DB*i* Manual
ONENTRY	install a JPL procedure or C function which JAM/DB*i* will call before executing a sql or dbms statement	JAM/DB*i* Manual
ONERROR	install a JPL procedure or C function which JAM/DB*i* will call whenever a sql or dbms statement fails	JAM/DB*i* Manual
ONEXIT	install a JPL procedure or C function which JAM/DB*i* will call after executing a sql or dbms statement	JAM/DB*i* Manual
PREPARE_COMMIT	prepare to commit a transaction	SYBASE Notes
ROLLBACK	rollback a transaction	SYBASE Notes
SAVE	save a two-phase commit	SYBASE Notes
SET	set execution behavior for a procedure (execute all, stop at fetch, etc.)	SYBASE Notes
SET_BUFFER	set up a buffer for engine–supported scrolling	SYBASE Notes
START	set the first row for JAM/DB*i* to return from a SELECT set	JAM/DB*i* Manual
STORE	store the rows of a SELECT set in a temporary file so that the application may scroll through the rows	JAM/DB*i* Manual
TRANSACTION	set the default transaction	SYBASE Notes
TYPE	set data types for parameters of a stored procedure executed with an rpc cursor	SYBASE Notes

Command	*Description*	*Documentation*
UNIQUE	suppress repeating values in a selected column	JAM/DB*i* Manual
UPDATE	update a table while browsing	SYBASE Notes
USE	open an existing database	SYBASE Notes
WITH CONNECTION	set the default connection for the duration of a command	JAM/DB*i* Manual
WITH CURSOR	specify the cursor to use for a statement	JAM/DB*i* Manual
WITH ENGINE	set the default engine for the duration of a command	JAM/DB*i* Manual

JAM/ ReportWriter

Release 5.1

November 12, 1993

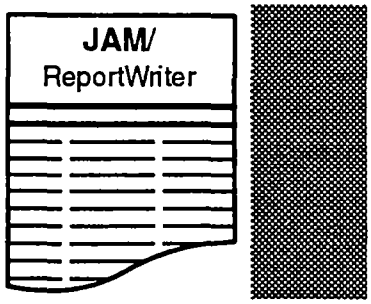
This is the manual for **JAM/ReportWriter** Release 5.1. It is as accurate as possible at this time. Both **JAM/ReportWriter** and this manual are subject to revision.

Please send suggestions and comments to:

Technical Publications Manager
JYACC, Inc.
116 John Street
New York, NY 10038

(212) 267–7722

TABLE OF CONTENTS



Chapter 1
Introduction

JAM/ReportWriter is an add-on to JAM and JAM/DB*i* that allows developers to define and produce complex reports and analyses for their JAM/DB*i* or stand–alone database applications.

JAM/ReportWriter is part of a family of JYACC products. The following table describes the rest of the family:

Product	*Description*
JAM®	JYACC Application Manager
JAM/DB*i*®	Interface for SQL relational database systems
JAM/P*i* for Motif	Presentation interface for the Motif GUI
JAM/P*i* for Microsoft Windows	Presentation interface for Microsoft Windows
JAM/P*i* for Graphics	Presentation interface for Graphics
Jterm®	Color Terminal Emulator optimized for JAM

This manual is intended for developers who are using JAM/ReportWriter for the first time and for more experienced developers who wish to gain a better understanding of this product.

To get the most out of JAM/ReportWriter and this manual, you should be familiar with both JAM and JAM/DB*i* and with the concepts discussed in the *Overview* portions of their respective manuals.

1.1 ABOUT THIS MANUAL

This manual is both a user's guide and a reference manual. Chapters 1 through 3 serve as a general introduction to JAM/ReportWriter, its features, and concepts. Chapters 4 through 10 provide instructions for developing reports, with Chapters 11 through 13 constituting the reference portion of the manual. In addition, four appendices provide further reference information such as a glossary of reserved words, implementation notes, a troubleshooting guide, and report development examples.

- Chapter 1, ***Introduction***, describes the organization of this manual, introduces ReportWriter terminology and concepts, and lists the typographical conventions used in the reference chapters.
- Chapter 2, ***ReportWriter Philosophy***, describes the product's features and explains how ReportWriter takes advantage of its integration with JAM and JAM/DB*i*. In addition, this chapter identifies the differences between release 5.1 and previous releases of ReportWriter.
- Chapter 3, ***Sample Application***, shows the development of two simple reports based on the sample application originally introduced in the JAM and JAM/DB*i* manuals. These examples are provided to illustrate the process of developing reports with JAM/ReportWriter.
- Chapter 4, ***The Report Format Screen***, explains how to create and compile the report format screen.
- Chapter 5, ***Using the Script Statements***, thoroughly describes each of the statements in the report scripting language.
- Chapter 6, ***Report Components***, provides additional information about variables, subreports, function calls, report areas, and queries.
- Chapter 7, ***Processing Flow***, consolidates the order–of–processing issues raised in other contexts in previous chapters. This chapter develops a generic example to illustrate these topics. The chapter also summarizes ReportWriter's default pagination rules and explains how the report developer can suppress their application.
- Chapter 8, ***ReportWriter Input and Output***, describes device configuration files, explains how ReportWriter resolves conflicting output specifications, and provides guidelines for writing custom input and/or output functions.

- Chapter 9, ***Running ReportWriter***, explains how to invoke ReportWriter—from the command line as a stand–alone application or from a JPL, C, or other supported language routine within a JAM/DB*i* application.
- Chapter 10, ***Developments Hints***, describes several techniques to help you to take advantage of ReportWriter's capabilities. Among these are sub–queries, end user control over report composition, and converting reports developed under ReportWriter release 4. Chapter 10 also identifies JAM features which are particularly useful in report development.
- Chapter 11 is the ***Script Statement Reference.***
- Chapter 12 is the ***Library Function Reference.***
- Chapter 13 is the ***Utilities Reference.***
- Appendix A is a ***Glossary of Reserved Words.***
- Appendix B, ***Implementation Notes***, provides information that will be of use to some report developers. The topics covered include: how to create a custom version of the report generation program executable, and how to use onscreen arrays as target variables for a database fetch.
- Appendix C, ***Troubleshooting***, provides a chart listing common problems in report composition and suggestions for correcting them.
- Appendix D, ***Examples***, illustrates, by means of sample reports, various techniques for using ReportWriter's capabilities.

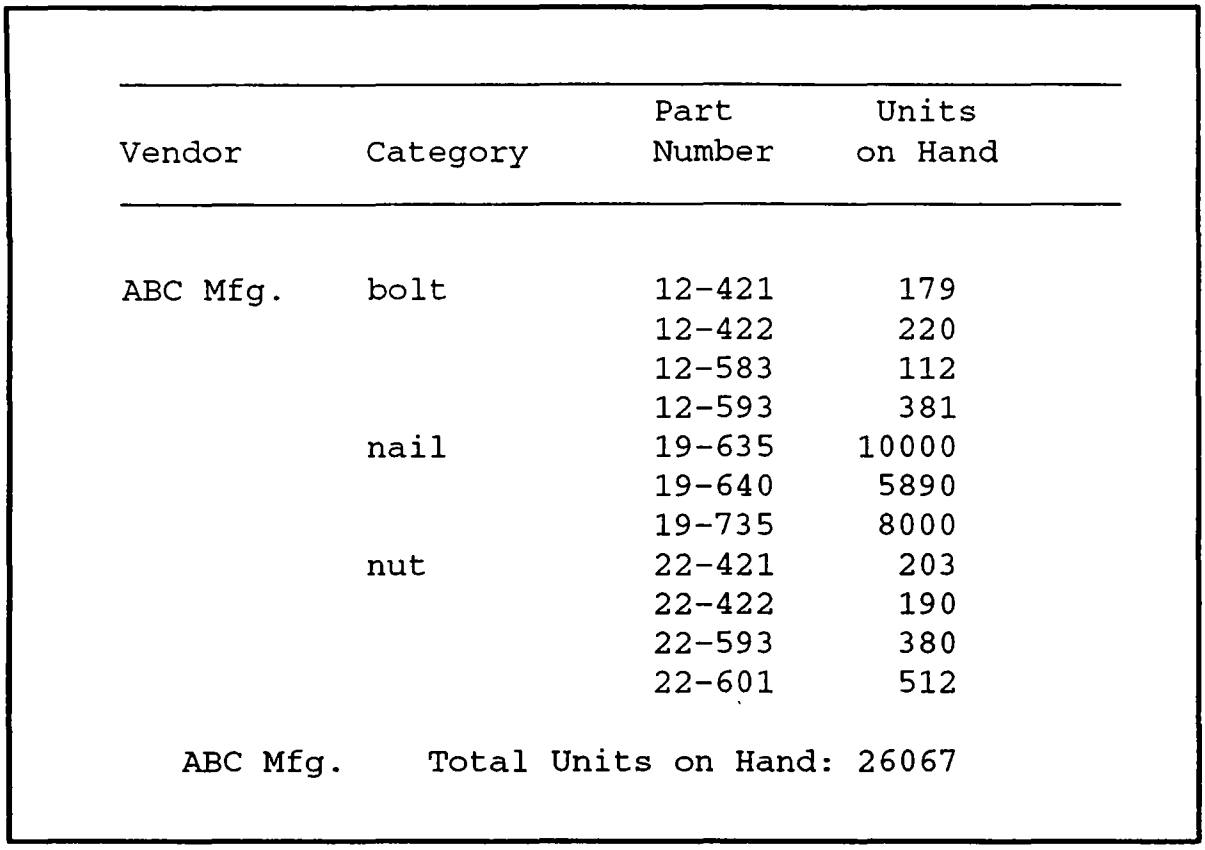
1.2 TERMINOLOGY

The following discussion introduces terminology and concepts used throughout this manual.

A *break* is a logical division in a report, whether between pages (a page break) or between groups of data (a data break).

As the term implies, a *page break* occurs at the end of each page of the report.

A *data break* occurs when there is a change in the value of one or more variables designated as *break fields*. A *break group* is the data associated with a particular value of the break field. Figure 1 shows a report in which `category` and `vendor` are defined as break fields; all entries for a given vendor constitute a break group; within the listing for a single vendor, all items in the same category consitute a lower–level break group.



Vendor	Category	Part Number	Units on Hand
ABC Mfg.	bolt	12-421	179
		12-422	220
		12-583	112
		12-593	381
	nail	19-635	10000
		19-640	5890
		19-735	8000
	nut	22-421	203
		22-422	190
		22-593	380
		22-601	512

ABC Mfg. Total Units on Hand: 26067

Figure 1: A Report With Break Fields

Throughout this manual, whenever the term *break* is used without qualification, it refers to a data break.

Both page and data breaks can have associated *headers* and *footers*. Header refers not only to the text printed at the beginning of a page or break group, but also to any processing invoked at this point. Similarly, a footer includes both the text output as well as any processing associated with the end of a page or break group.

Page headers and footers are distinct from break headers and footers.

The *page header* and *page footer*, if defined for the report, occur at the beginning and end, respectively, of each page and are independent of the report data on the page. Page headers or footers typically include some processing to calculate and display the page number, and might show a title for the report or report section.

Break headers are likely to display column headings or some kind of identification for the break group that follows and may include processing to reinitialize variables such as running totals for the break group. *Break footers* are often used to output subtotals for the break group and to update running totals for the next higher break level or for grand totals.

The format for a report is defined on a *report format screen.* This is a **JAM** screen that contains the formatting information for any or all report areas. A report *area* is any separately identifiable section of the report: a page header, a title page, the format for reporting data retrieved from the database, a break footer, etc.

In this manual, the term *area* is used in reference both to the actual contents of a finished report as well as to these components defined and labelled in the report format screen.

The *detail* area of a report is driven by a database query and consists of the processing and output associated with each row retrieved from the database in response to the query. Data break checking and processing take place within the context of processing for the detail area. A report format screen can contain more than one detail area, each with its own associated database query, processing, and output format. Each detail area defined in the report format can, of course, produce multiple copies of itself in the finished report—the number depending on how many rows are fetched and reported as a result of the query for that detail area.

1.3
A NOTE ABOUT LANGUAGES SUPPORTED

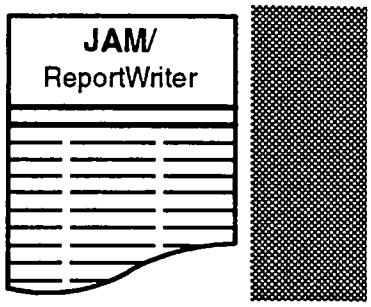
Throughout this manual, wherever reference is made to developer–written C functions, it should be understood that this applies also to any other language supported by the **JAM** and **JAM/DB***i* installation at your site. Sometimes the phrase "C (or other supported language)" is used as a reminder; the absence of this phrase should not be taken to exclude any other programming language you use to develop your **JAM** and **JAM/DB***i* applications.

1.4
CONVENTIONS USED

The following typographical conventions are used in the reference chapters of this manual:

`literal` This font is used for text that should be typed verbatim, for examples, and for file and directory names.

italics	Bold italics show where screen, file, and variable names should appear. Replace these with the appropriate names in your applications.
[*x*]	In this notation, the brackets indicate that *x* is an optional element. Do not type the brackets.
x...	Ellipses indicate that the element *x* may be repeated one or more times in the indicated location.
[*x*] *	An asterisk following an element indicates that the element can appear multiple times within a statement or clause. Do not type the asterisk.
[*x* | *y* | *z*]	The elements shown are mutually exclusive. No more that one may be present in the statement, clause, or command. Do not type the brackets or the vertical bars.
{*x* | *y* | *z*}	At least one of the elements shown must be present in the statement or clause. Unless otherwise indicated, more than one element in the group can appear. Do not type the braces or the vertical bars.



Chapter 2
ReportWriter Philosophy

JAM/ReportWriter is a report–generation tool that allows developers to design and create query–based reports for JAM/DB*i* or stand–alone database applications. Like the other products in the JAM family, it conforms to the JAM philosophy of platform– and database–independence.

2.1
HOW REPORTWRITER WORKS WITH JAM

JAM/ReportWriter, as an add–on to JAM and JAM/DB*i*, relies on JAM's functionality to define and produce reports. In particular, it utilizes the following JAM and JAM/DB*i* components:

- screen editor – to define the content and appearance of each section of the report,
- JPL, the JYACC Procedural Language – to direct any specialized processing or computations associated with the report, and
- JAM/DB*i* – to support the database query associated with each detail area and any additional database commands issued from the JPL or C routines associated with the report.

Each report is defined as a single JAM screen containing the formatting and JPL code for all report areas. The report script, which ties together all these components, is maintained in the screen JPL module.

Just as the developer of a JAM application uses the screen editor to paint each screen in the application, the report designer uses the screen editor's layout, data formatting, and editing capabilities to define the appearance and content of each report area (detail, page headers and footers, break headers and footers, title page, trailer, etc.) .

Since all report areas are drawn on a single screen, the developer can easily determine if related areas (such as column titles in the break header and the corresponding data in the detail area) will line up correctly when the report is printed. The report designer has just one screen to edit and one JPL module to maintain during the development process. The use of a single screen also minimizes or eliminates the need to use the LDB (Local Data Block) for passing report data at runtime.

In addition, report designers can take advantage of JAM's data formatting and editing capabilities, including

- a wide variety of date, time, and currency formats, including the ability to define custom currency formats,
- computations and numeric formatting,
- justification, and
- word wrap.

JPL procedures required for report processing are entered as named procedures in the screen JPL module.

Report areas and associated processing, thus defined, are linked together through the ReportWriter script to generate a report.

The ReportWriter script language is designed to take advantage of JAM's data handling capabilities. This not only provides for access to the data being reported, but also allows for flexible report definition. Through the use of colon–expanded variables, for example, the selection of report components, header and footer text, query arguments, and the like, can be controlled by data available at runtime, whether the result of a database query, computations performed in JPL or C language routines, or user input.

JAM/DB*i* capabilities are also available to the report developer. The report script provides a mechanism for issuing SQL (Structured Query Language) statements to fetch data from the database. The report developer can access all other JAM/DB*i* capabilities from JPL procedures and C (or other supported language) routines invoked through the report script.

2.2
A DEVELOPER'S TOOL

Note that while end users can initiate report generation, either from within a JAM/DB*i* application or by entering the appropriate command at the system prompt, report definition is performed by JAM developers within the authoring environment, as distinguished from the production environment. JAM/ReportWriter is not intended to allow users to construct reports "on the fly."

Developers can, however, define flexible report formats that are sensitive to parameters entered by the end user. While this allows end users to have a certain degree of control at runtime over the content and format of the report, it is the developer, not the end user, who is responsible for constructing the various formats that may be used and for creating a report script that can respond to specified user input.

2.3
FEATURES

2.3.1
Fully Integrated with JAM and JAM/DB*i*

Since JAM's standard tools are used to create reports under JAM/ReportWriter, developers can leverage their existing knowledge of JAM and JAM/DB*i*: text and data are placed and formatted by creating standard JAM screens; information is retrieved from the database using JAM/DB*i*'s interface; and control and manipulation are provided through JPL or C subroutines. The only new component the developer must learn is the report script, which ties the pieces together to create the report.

JAM variables communicate information between JAM/ReportWriter and the JAM/DB*i* application. This provides considerable flexibility in report composition. For example, report breaks can be defined on any JAM field, whether fetched in the report query or computed in a JPL or C routine. In addition, using JAM's colon expansion feature, various report specifications, such as the `where` clause of the database query, can use a variable whose value is not known until runtime or can change as the application runs.

JAM/ReportWriter can be linked with any current version of JAM, JAM/DB*i* (including 4.x as well as 5.x), or JAM/P*i*. It supports all the features of any linked–in product, including intelligent colon expansion in JAM/DB*i* 5.x.

2.3.2

Support for Both Linked–in and Stand–alone Reports

Reports can be invoked from within the JAM/DB*i* application or directly from the command line. In addition, the calling sequence for invoking JAM/ReportWriter can be customized, if desired. (The source code for the Report Writer main program is supplied, allowing developers to make this and other modifications as needed.)

2.3.3

Databases Supported

ReportWriter can be used with all relational databases supported by JAM/DB*i*. It provides a single, consistent report generation tool across all supported databases.

In addition, developers can write their own row–supply functions, allowing data input from flat files and other sources. (Refer to Section 8.3 for further information on developer–written input functions.)

2.3.4

Support for Multiple Report Types

With JAM/ReportWriter, developers can produce reports of almost any type: columnar reports, statistical summaries, financial analyses with as many calculations, subtotals, and break levels as needed, or executive summaries showing just the summary data without the related detail.

JAM/ReportWriter reports can be grouped into the following general types:

Row–level report: In a row–level report, a detail line (or more) is printed for each row selected from the database. JAM/DB*i* Report Writer supports up to 20 break levels so that the report can also include subtotals or other results from the groupings (and sub–groupings). Row–level reports are driven by database queries (or fetches from alternate input sources).

Summary report: Like row–level reports, summary reports perform computations based on rows fetched from the database or other sources.

In summary reports, however, only the results of the computations, not the fetched data, are printed.

Instance report: Any tagged area on the report format screen can be printed as a report or a section of a report. Unlike a row–level report, an instance report is not generated repetitively as the result of a query, but, rather, is initiated by a statement that is executed once as it is encountered in the report script.

Multi–section report: Any of the above report types can be combined and paginated as a single, continuous report. The combination of a header page identifying the report, the main body of the report based on one or more database queries, and a trailer page containing grand totals is an example of a multi–section report. The header and trailer are instance reports; the body comprises one or more row–level reports.

2.3.5

Intelligent Page Break Control

JAM/ReportWriter automatically chooses page breaks that are reasonable and visually appealing. For example, break headers are not printed at the bottom of a page, and forms are not unnecessarily split across pages. Alternatively, report designers can override these options.

Of course, the report designer can also specify all page breaks related to report content, at each detail row, for instance, or at various break levels.

2.3.6

Non–Procedural Report Script

The report script is written in a high–level, non–procedural language and is maintained in the report's screen–level JPL module.

Report generation is an event–driven procedure: the script indicates the appropriate action to take when some specified event (such as a change in the value of a particular variable) occurs. The report script, in effect, orchestrates the various report components while remaining independent of their details—formatting information is contained in each area of the JAM report screen, and any arithmetic or other processing is contained in JPL (or C, etc.) routines. Because the report script is insulated from such details, report designers can make changes to report formatting and calculations without the need to edit the script itself.

2.3.7

Dynamic Report Composition

A single report definition will, of course, produce different reports if given different data—different data printed, different page sizes, breaks at different places, etc.

JAM/ReportWriter, however, also allows developers to define a report in which the report definition, itself, can vary under the control of a JAM application. Through the use of colon–expanded JAM variables in the report script, the following information can be supplied to the ReportWriter at runtime:

- arguments to the database query,
- field(s) to break on in row–level and summary reports,
- the report area to be output, and
- JPL, C, or other supported language routines to be invoked.

For example:

1. In a report of account information, it might be useful to let the end user specify the saleperson whose accounts should be listed. The application that generates the report would include a prompt for the user to enter this information. In the report script, the `where` clause of the database query would be written with the appropriate colon–expanded variable, so that the requested salesperson's ID will be inserted into the SQL statement at runtime.
2. The type of information in a report containing municipal data might vary, depending upon whether it relates to a city, a town, or a village. The report developer would create three different report areas for detail information; the area selected for any particular row would be determined by the value of the variable that identifies the means of government for the municipality represented by that row.

2.3.8

Device–Specific Processing

ReportWriter is output device– and printer–independent, generally relying on the operating system to drive the printer or other output.

Developers can, however, implement device support through the mechanism of device configuration files. Any or all of the following information can be entered into these files:

- initialization and reset strings,
- page width and length,
- left margin,
- output device, file name, or process to which output is spooled, and
- name of a developer–written output function and the size of its output buffer.

The applicable device configuration file, if any, is specified when the JAM/ReportWriter is invoked.

2.4
NEW FEATURES IN JAM/ReportWriter RELEASE 5

Developers familiar with earlier versions of ReportWriter will notice the following differences in upgrading to release 5:

2.4.1
A Single Report File

Under ReportWriter 5, the entire report specification is contained in a single file.

Instead of separate screens for each report area, the layouts for all areas are consolidated on a single screen. Each line of the screen is tagged with the name of the report area(s) to which it applies. These name tags correspond to the form names referenced in the `form` clauses of earlier report scripts.

The report script is also contained in the report format screen. It appears as comments in the screen JPL module. All JPL procedures used in the report may also be included in the screen JPL module.

Consolidating the report into a single file provides the following advantages over the previous method of using a separate file for each JPL procedure and a separate JAM screen for each report area:

- Greater runtime efficiency: ReportWriter need not continually open and close files during report generation.

- Reduced need to use the LDB for passing data: Since all report areas are contained on a single screen, field data from one report area can be propagated to any other area or used by any JPL or C routine without need of passing it through the LDB.
- Ease of aligning report areas: By placing all report areas on the same screen, the report developer can readily line up related fields from different report areas, such as aligning columns in the detail area with the corresponding headings in the associated break header or with totals fields in a break footer.

2.4.2
Modularity

Report developers can still maintain common modules for report components (source screens and JPL code) that are shared among reports.

ReportWriter release 5 has a new `include screen` compiler directive that causes the compiler to include specified source screens in the compiled (binary) report. Thus, a common page header, for example, need be defined only once and can be shared by any number of reports.

JPL continues to be accessible from libraries or separate files, as well as from the report format screen.

Refer to Section 4.3.2 for more information about the `include screen` compiler directive.

2.4.3
Field Name Aliasing

Report design will at times require the same field to appear in more than one report area. ReportWriter release 5 provides a field–naming syntax that allows developers to equate field names on the report format screen while still observing JAM's restriction against duplicate field names on a single screen.

When the `include screen` compiler directive is used, ReportWriter automatically aliases and equates fields on the included screen to the identically–named fields on the report format screen and to identically–named fields on other included screens.

Refer to Section 4.1.3 for more information about ReportWriter's field naming conventions.

2.4.4

Compatibility with Release 4 Reports

Release 5 of ReportWriter includes the utility `rw4to5`, which converts reports developed under ReportWriter release 4 into the single–file format required for release 5. This utility consolidates the screens, JPL modules, and report script file of the release 4 report and produces a release 5 report format screen with area tags added and field name aliases provided where needed.

Refer to Chapter 13 for detailed information on this utility.

2.4.5

Append Output Option

When ReportWriter is invoked iteratively from within a JAM/DB*i* application, the output of successive invocations can be consolidated into a single, through–paginated report.

A new command line option, `-a`, directs ReportWriter output to be appended to the specified file. This option allows an application to invoke ReportWriter in a loop, without overwriting output from previous iterations.

Refer to Sections 9.1 and 9.4 for further information on the use of this output option.

2.4.6

Finer Control Over Break Processing

Report developers can now control where break processing should occur in the flow of detail processing. Put the keyword `breakcheck` in the `detail` statement at the point where you want break processing to take place. If this keyword is omitted, break processing is done immediately before the detail form is output or, if no form is specified, after all `jpl` and `call` clauses are executed.

Refer to Section 5.1.2 for more information on the `breakcheck` keyword.

2.4.7

Row–Supply Hook Functions

ReportWriter uses JAM/DB*i* to fetch rows from the database. In some circumstances, however, developers might need to supply their own data from flat files, wire services, or other sources not supported by JAM/DB*i*.

In release 5, programs that fetch or generate such data can be linked with ReportWriter. The query specified in the `detail` statement is passed to the installed developer–written function rather than to JAM/DB*i*.

Refer to Section 8.3 for information on writing and installing developer–written row–supply functions.

2.4.8

Table–to–Report Utility

A new utility, `tbl2r`, creates a simple report format screen and report script to output the values in a database table. This report can also be used as a template or initial model for a more elaborate report based on the specified database table. (This utility must be linked with the JAM/DB*i* release 5 libraries.)

2.4.9

C–Style Comments in the Report Script

With release 5 of ReportWriter, comments in the report script are entered as they would appear in the C programming language: `/* comment */`.

Refer to Section 4.2.2 for a discussion of the report script format and syntax.

2.4.10

`area` Clause

The new reserved word `area` has replaced the ReportWriter release 4 reserved word `form`. `area` clauses and subclauses serve the same purpose in ReportWriter 5 scripts as `form` clauses and subclauses served in release 4.

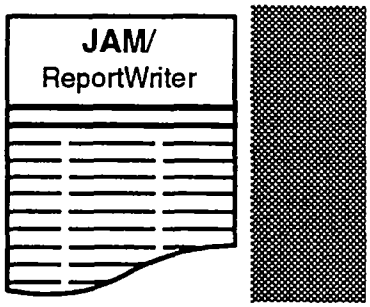
2.4.11
Improved "Shrink" Processing

A new method for shrinking report output areas has been implemented under ReportWriter release 5. Shrink processing in this release is geared specifically to the requirements of report production, condensing report areas vertically only and leaving intentional blank lines unaffected. As in ReportWriter release 4, use of the shrink feature for any report output area remains optional and is controlled by the report developer through the report script.

2.5
NEW FEATURES IN JAM/ReportWriter RELEASE 5.1

Developers familiar with ReportWriter release 5.0 should note that the following features have been added to release 5.1:

- full subreport capability
- argument passing on invocation of reports and subreports
- multiple areas and subreports per statement
- named cursors
- variable substitution for numeric values
- use of array elements as break fields



Chapter 3
Quick Start and Sample Application

This chapter illustrates the procedure for creating reports with **JAM**/ReportWriter. The first section summarizes the steps you would use to develop and generate a report. The remainder of this chapter develops a report based on the sample application introduced in the **JAM** *Development Overview* and expanded upon in the **JAM**/DB*i* *Overview.*

The example in this chapter is not intended to be a comprehensive demonstration of ReportWriter's features. Rather, it builds a modest report based on a fairly simple database application. This approach will allow you to follow the basic procedure without getting bogged down in the intricacies of a major application or a complex report.

This section begins with a summary of the application and its associated database tables. If you want to review the sample application in more detail, refer to the manuals mentioned above.

Following the application description, a report is presented. This report retrieves data from two database tables, consists of several different report areas—a title page and a break footer, as well as a detail area—and uses data breaks to organize the detail output.

This chapter explains the steps required to create the report and shows the report format screen layout and associated JPL.

Appendix D, "Examples," shows how to use ReportWriter's capabilities to extend this basic example to meet more complex reporting needs.

3.1 QUICK START

This section summarizes the steps for developing and generating a report with JAM/ReportWriter.

Depending upon your application, you may choose to create the report format screen yourself or to use the `tbl2r` utility to create a basic report format screen and a rudimentary report script from a database table.

1. Create the report format screen.

   Using any JAM screen editor (`jxform`, `jxdbi`, or `jxrw`), create a report format screen that includes the areas you want in your report. Refer to Section 4.1 for an explanation of the report format screen.

   Or

   Run the `tbl2r` utility to create a basic report format screen from a specified database table. Refer to Chapter 13 for a detailed description of this utility. A basic report created with `tbl2r` is shown in Appendix D, Section D.2.1.

2. Enter the report script.

   Type the script into the screen JPL of your report format screen. The report script is described in Section 4.2; the script statements are covered in detail in Chapters 5 and 11.

   Or

   If you have used the `tbl2r` utility to create the report format screen and its associated script, you can edit the screen and the script as needed. Section D.2 shows a report created with this utility and subsequent modifications.

3. Save the report format screen.

4. Compile the report.

   Use the `rprt2bin` utility to compile the report format screen. This procedure is described in Section 4.4. The `rprt2bin` utility is described in detail in Chapter 13.

5. If you need to specify non–standard output parameters:

   Create and compile a device configuration file.

Device configuration files are described in Section 8.1; they are compiled with the `dev2bin` utility, which is covered in Section 8.1.3 and in Chapter 13.

6. Run the report.

   You can invoke ReportWriter

   - from the command line with the `rwrun` utility,
   - from a JPL procedure in your application* with the `rwrun` JPL command, or
   - from a C routine with the `dbi_rwrun` library function.

   Refer to Chapter 9 for more information on running ReportWriter. The `rwrun` utility is described in Chapter 13; the `dbi_rwrun` library function is covered in Chapter 12.

* JAM/ReportWriter can be invoked only from `jxrw` or `jamrw`, the JAM authoring and application executables, respectively, with ReportWriter linked in. It cannot be invoked from `jxform`, `jam`, `jxdbi`, or `jamdbi`, as ReportWriter is not linked into these executables.

3.2
REVIEW OF THE SAMPLE APPLICATION

The sample application presented in the JAM and JAM/DB*i* manuals is the personnel portion of a human resources application. It consists of a series of screens for the end user to log on to the system, select the desired application, and, in the `Personnel` application, enter or review employee information. These screens are shown in Figure 2 through Figure 4.

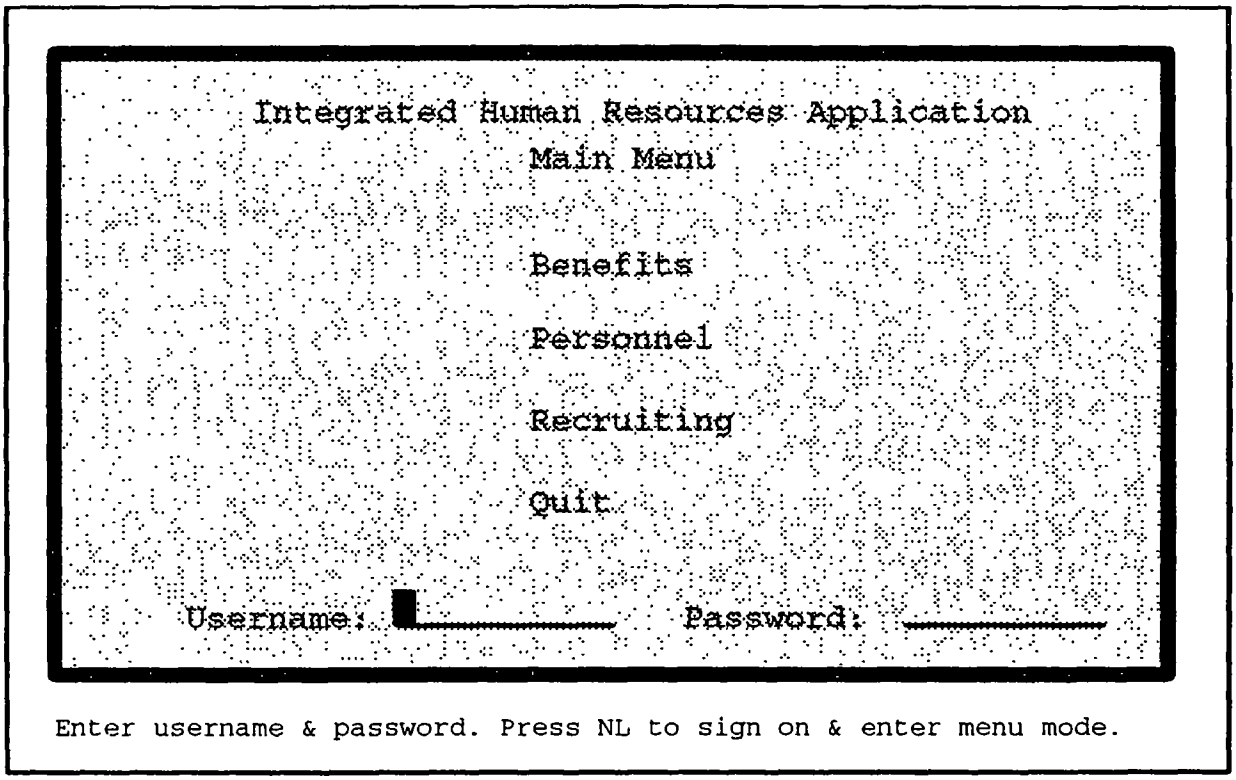


Figure 2: Human Resources Application Main Menu `mainscrn`.

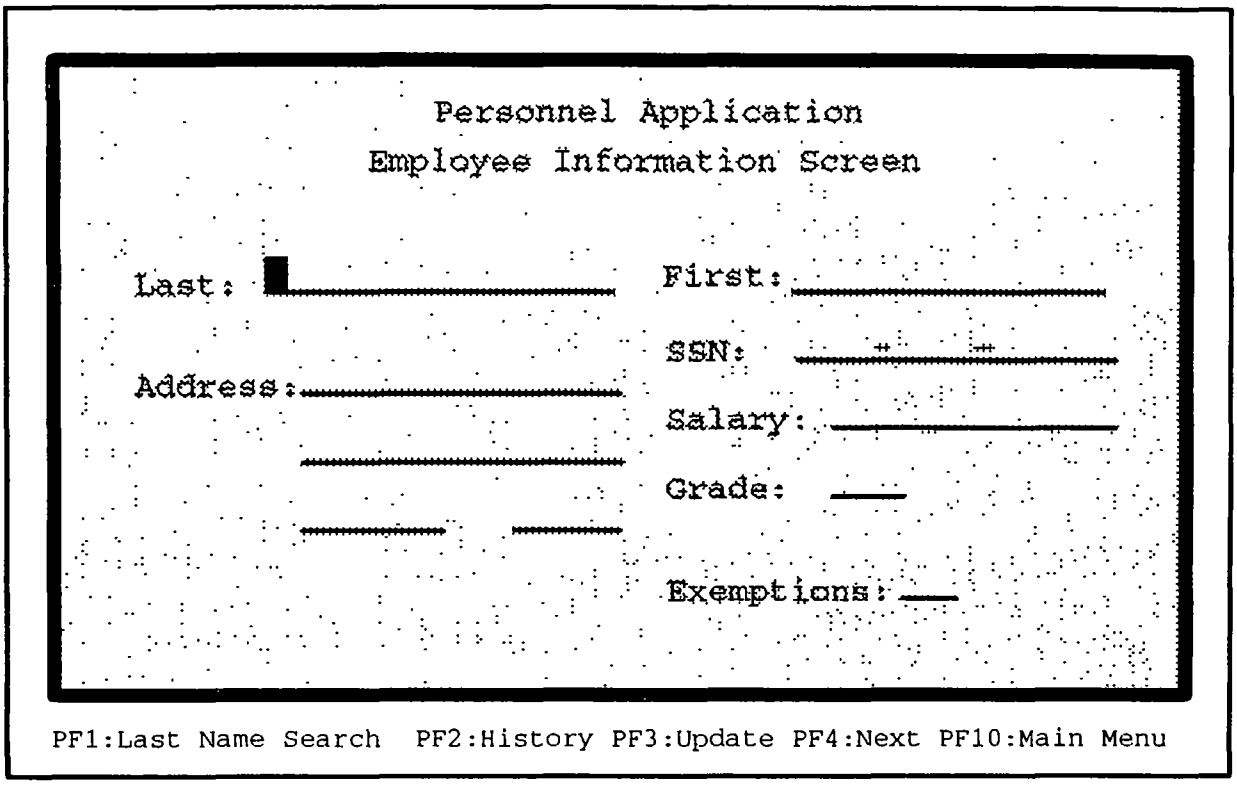


Figure 3: Personnel Application Employee Screen `empscrn`.

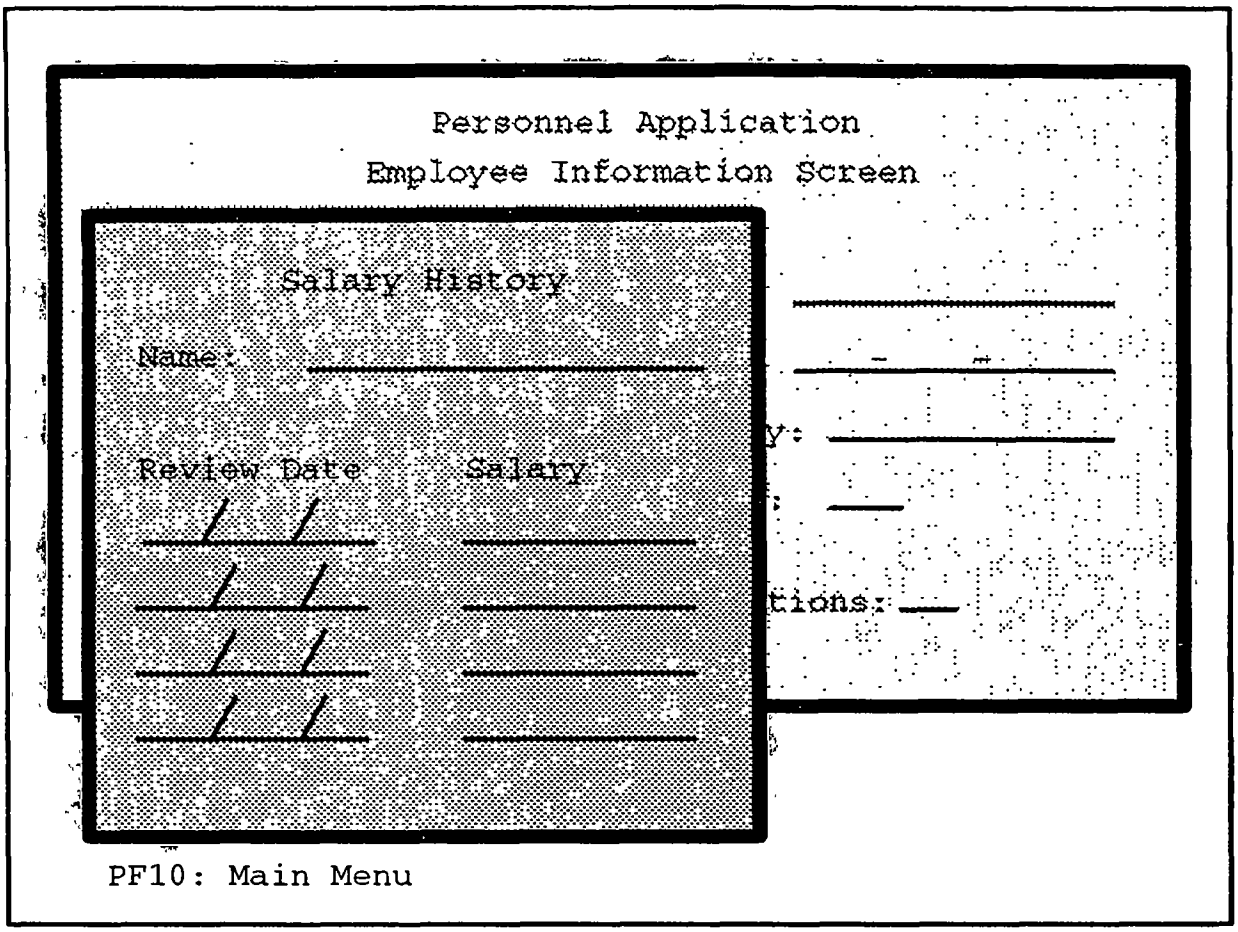


Figure 4: Personnel Application Salary History Window `salhist`.

The personnel application relies on three database tables: `emp`, `acc`, and `review`. These tables, plus sample SQL statements for creating them, are shown below. (Note that some database engines use different names for column datatypes.)

The table entries represent seven employees.

Table emp has eight columns. Each row stores an employee's social security number, name, home address, and current grade.

```
CREATE TABLE emp (
   ssn       CHAR(11)        NOT NULL,
   last      CHAR(20),
   first     CHAR(12),
   street    CHAR(20),
   city      CHAR(15),
   st        CHAR(2),
   zip       CHAR(5),
   grade     CHAR(1))
```

ssn	last	first	street	city	st	zip	grade
038–68–6826	Jones	Barnabus	321 West 11 St	Albuquerque	NM	87124	C
122–98–6541	Aumond	Hilary	11–12 Front St	Albuquerque	NM	87124	E
122–99–4102	Jones	Michael	5 Maple Drive	Albuquerque	NM	87124	B
139–42–1651	Blake	Norman	34 Concord Ave	Albuquerque	NM	87124	D
154–32–6610	Cory	Richard	411 Ann St	Albuquerque	NM	87124	D
310–77–3997	Grundy	Janet	70–2 Poe Ave	Albuquerque	NM	87124	D
310–32–0084	Jones	John P	9 Vern Terrace	Albuquerque	NM	87124	D

Figure 5: Table emp

Table acc has three columns. Each row stores an employee's social security number, current salary, and a number of tax exemptions.

```
CREATE TABLE acc (
   ssn       CHAR(11)        NOT NULL,
   sal       NUMERIC(10.2),
   exmp      NUMERIC(1))
```

ssn	sal	exmp
038–68–6826	29500.00	1
122–98–6541	37800.00	3
122–99–4102	26000.00	3
139–42–1651	89500.00	2
154–32–6610	43100.00	4
10–77–3997	38000.00	1
310–32–0084	47500.00	5

Figure 6: Table acc

Table `review` has four columns. Each row stores an employee's social security number, a hire date or review date, a new salary if it has changed since the previous review, and a new grade if it has changed since the previous review. If `newsal` or `newgrade` is null, the employee was reviewed but there was no change in salary or grade.

```
CREATE TABLE review (
   ssn       CHAR(11)          NOT NULL,
   revdate   DATE              NOT NULL,
   newsal    NUMERIC(10.2),
   newgrade  CHAR(1))
```

ssn	revdate	newsal	newgrade
038–68–6826	12/13/90	49500.00	C
038–68–6826	12/11/89	45000.00	NULL
038–68–6826	12/15/88	NULL	NULL
038–68–6826	12/14/87	38500.00	D
122–98–6541	04/10/90	37800.00	NULL
122–98–6541	04/08/89	31000.00	E
122–99–4102	05/01/90	29000.00	NULL
122–99–4102	05/01/89	25200.00	E
139–42–1651	11/12/90	89500.00	NULL
139–42–1651	11/08/89	81000.00	B
139–42–1651	11/10/88	67500.00	C
139–42–1651	11/10/87	NULL	NULL
139–42–1651	11/08/86	53000.00	D
154–32–6610	02/01/91	43100.00	D
310–77–3997	07/16/90	38000.00	D
310–77–3997	07/14/89	30000.00	E
310–32–0084	03/01/91	47500.00	D
310–32–0084	03/01/90	43000.00	E

Figure 7: Table `review`

3.3

ADDING A REPORT TO THE SAMPLE APPLICATION

Figure 8 shows a report based on the sample application. This report and its components are described in the following sections.

NOTE: In some distributions of JAM/DB*i*, the Social Security Number columns in the sample database tables were labeled `ss` instead of `ssn`, as shown in Section 3.2. The code shown in the examples in this section is based on this column being named `ssn`.

3.3.1

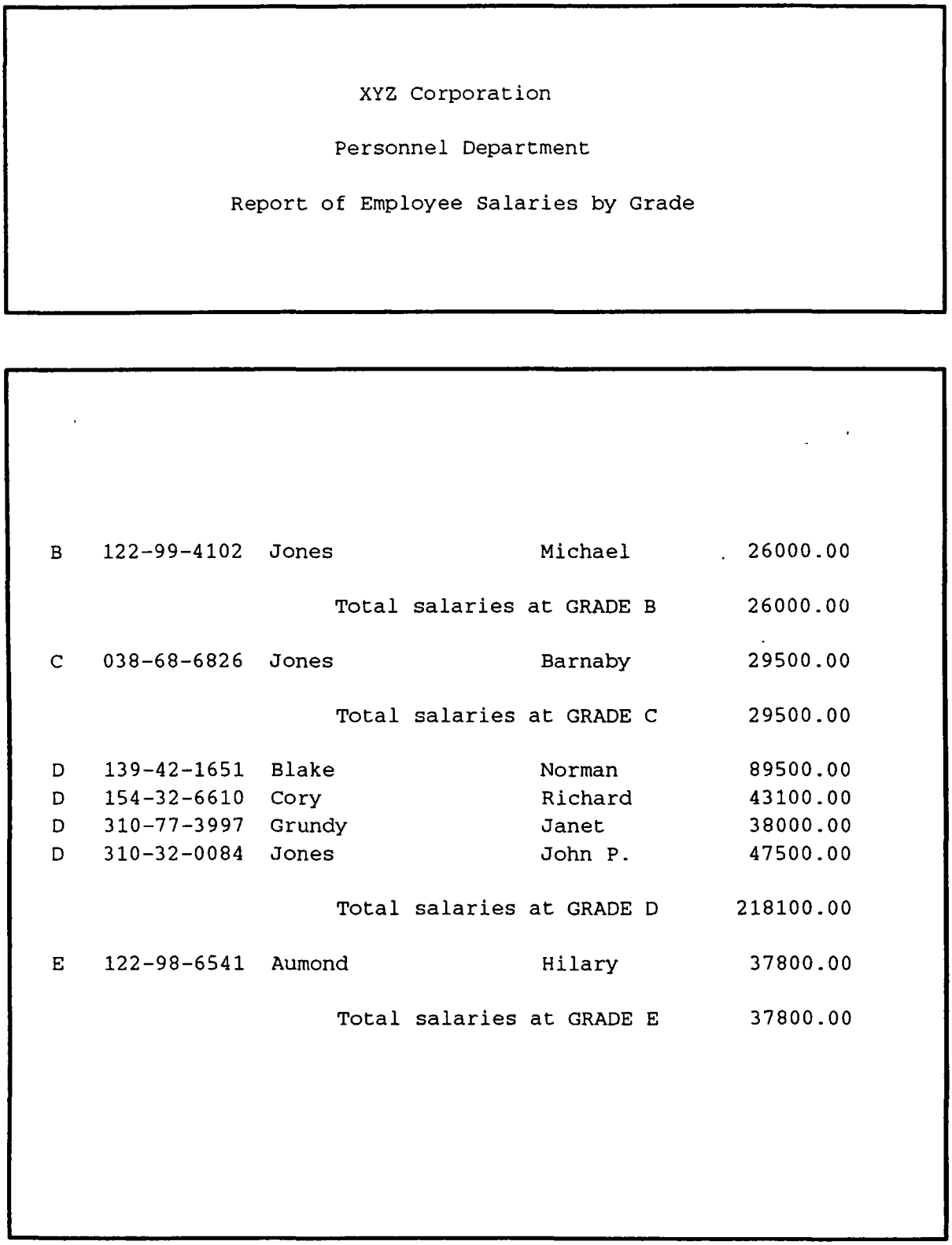
User's View

The report shown in Figure 8 is a listing of employees and their salaries. The report begins with a title page followed by a detail area that lists the employees alphabetically within employment `grade`. The salary total for each grade is shown.

3.3.2

Developer's View

The report format screen for this report is named `sal_rpt.jam`. Figure 9 shows the screen JPL module for `sal_rpt.jam`; it includes both the report script as well as the JPL procedures.



XYZ Corporation

Personnel Department

Report of Employee Salaries by Grade

B	122-99-4102	Jones	Michael	26000.00
		Total salaries at GRADE B		26000.00
C	038-68-6826	Jones	Barnaby	29500.00
		Total salaries at GRADE C		29500.00
D	139-42-1651	Blake	Norman	89500.00
D	154-32-6610	Cory	Richard	43100.00
D	310-77-3997	Grundy	Janet	38000.00
D	310-32-0084	Jones	John P.	47500.00
		Total salaries at GRADE D		218100.00
E	122-98-6541	Aumond	Hilary	37800.00
		Total salaries at GRADE E		37800.00

Figure 8: Output of Salary Report

```
# << begin report >>
#
# init        jpl    = startup     /* invoke the JPL proc
#                                     "startup" */
#             area   = titlepg     /* output the report
#                                     title page */
#             newpage              /* ensures that next
#                                     output after title
#                                     page begins on a new
#                                     page */
#
# break       field  = grade       /* group report output
#                                     by employment
#                                     grade */
#             footer area   = bfoot
#                     jpl   = reset_tot  /* after break
#                                     footer area is
#                                     output, invoke JPL
#                                     procedure to reset
#                                     the cumulative
#                                     salary total */
#
# detail query = "SELECT emp.ssn, emp.last, \
#                    emp.first, emp.grade, acc.sal \
#                    FROM emp, acc \
#                    WHERE emp.ssn = acc.ssn \
#                    ORDER BY emp.grade, emp.last, \
#                                emp.first"
#         area  = employee
#         jpl   = add_salary       /* invoke JPL procedure
#                                     to maintain running
#                                     total of salaries for
#                                     the current break
#                                     group (grade) */
#
# insert      jpl    = cleanup
#
# << end report >>
```

(figure continued on next page)

Figure 9: Screen JPL Module for `sal_rpt.jam` – Report Script

```
proc startup
       # This procedure is invoked once, at the start of
       # report generation. It performs two functions:
       #      1) initializes variables used in the report
       #      2) opens a connection to the database

       # the following statement initializes the running
       # total of salaries for the break group (grade)
  cat sal_tot

       # Since this report will be run stand-alone, we
       # need to open a database connection from within
       # ReportWriter. If the report were invoked from
       # within a DBi application, the application would
       # usually handle opening and closing connections.

       # The following syntax is specific for Sybase:

  dbms DECLARE rw_conn CONNECTION FOR USER user \
       PASSWORD pword


proc reset_tot
       # After the break footer has been output, the
       # cumulative salary total must be reset for the
       # next break group

  cat sal_tot


proc add_salary
       # Add the salary for this employee to the
       # running total of salaries at this grade

  math sal_tot = sal_tot + sal


proc cleanup
  dbms CLOSE CONNECTION rw_conn
```

Figure 9 (cont'd.): Screen JPL Module for `sal_rpt.jam` – JPL Procedures

Figure 10 is the report format screen layout. Each line of the layout is tagged to indicate the report output area it belongs to. The report output is reproduced again in Figure 11 to show which part of the output corresponds to which area.

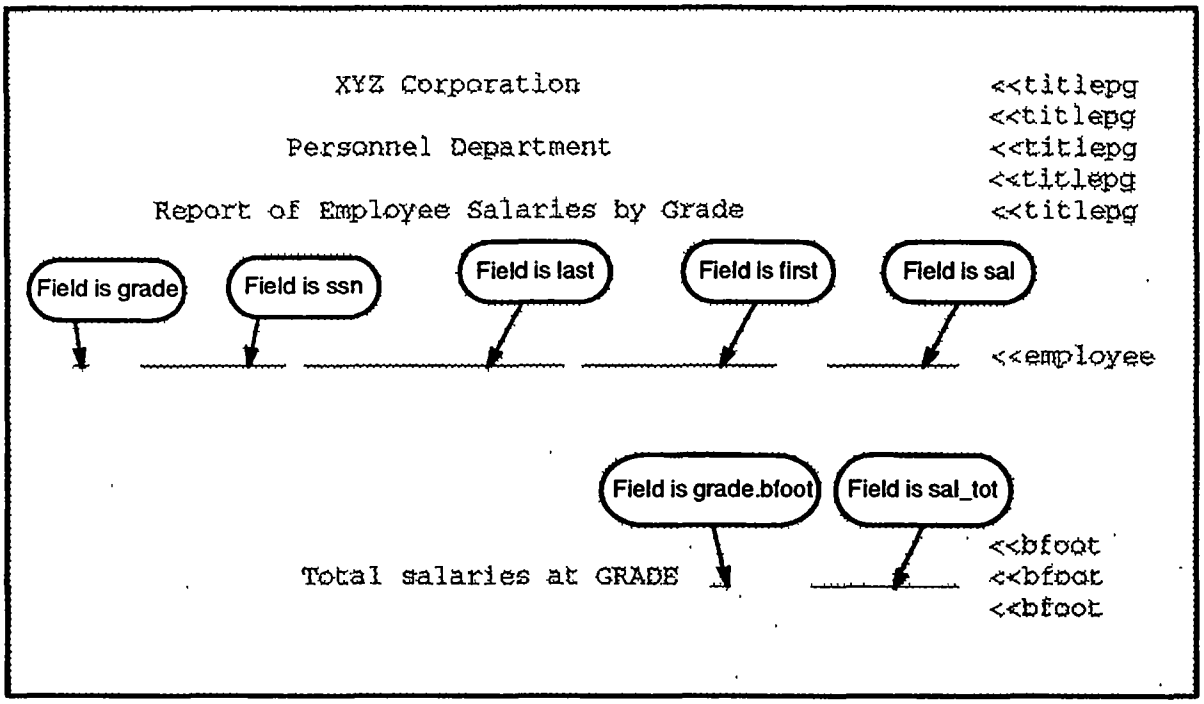


Figure 10: Report Format Screen `sal_rpt.jam` – Output Areas

This report must be compiled with the `rprt2bin` utility:

```
rprt2bin sal_rpt
```

The report generation program is invoked, either as a stand–alone utility (as shown in this example) or from within a JAM/DB*i* application, to produce the finished report. The command used to generate the report shown here is

```
rwrun sal_rpt
```

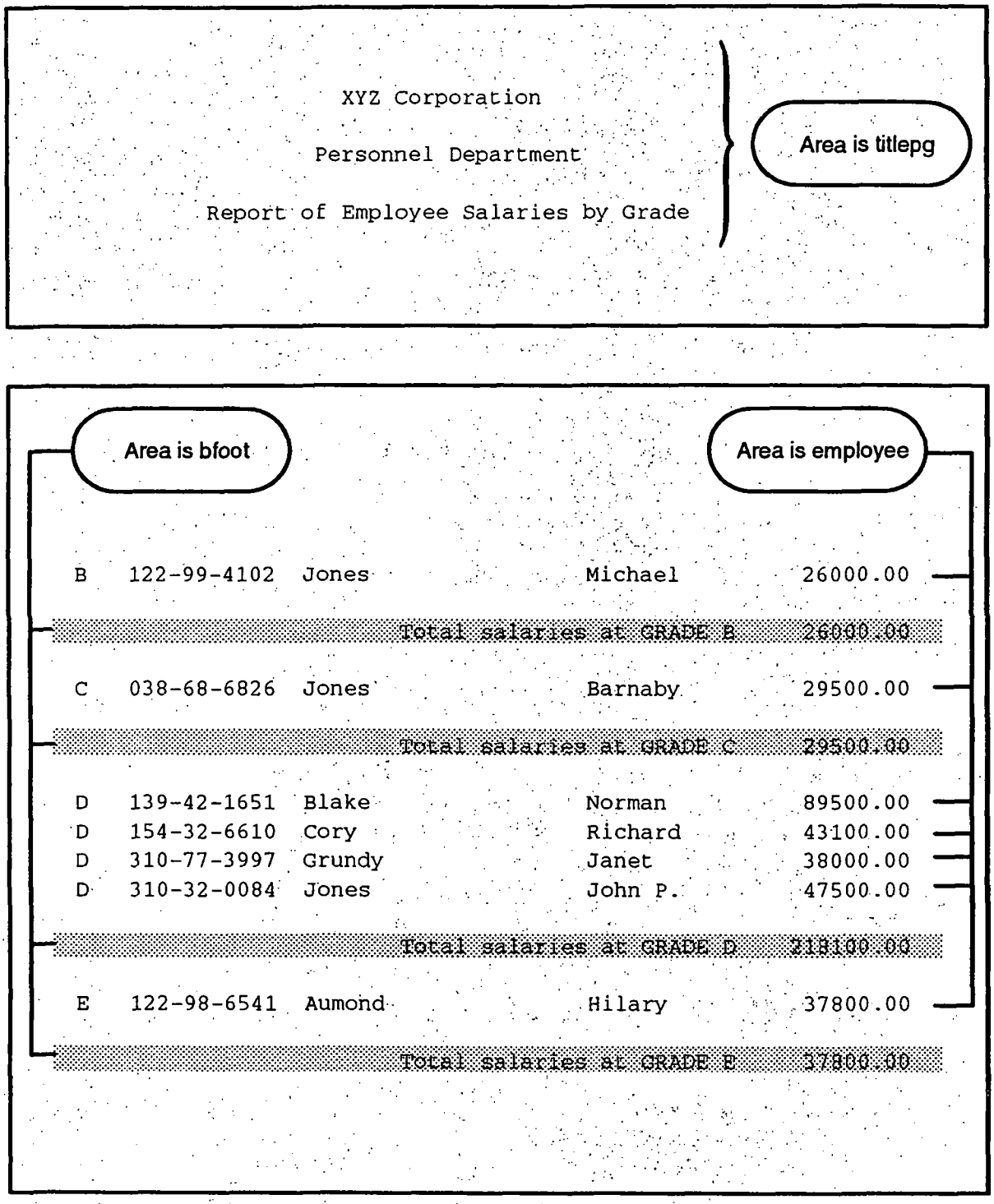
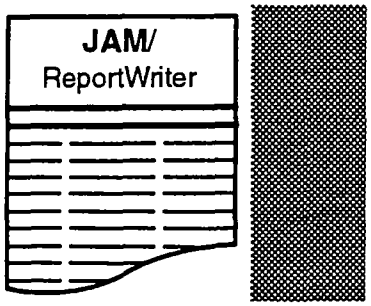


Figure 11: Output of Report `sal_rpt` with Corresponding Report Areas Indicated



Chapter 4

The Report Format Screen

Each report is defined as a JAM screen containing the formatting and JPL code for all report areas. The report script, which ties together all these components, is also maintained as part of the report format screen—as comments in the screen JPL module.

This chapter explains how to create the report format screen.

Section 4.1 explains how to layout and identify the report areas on this screen.

Section 4.2 describes the syntax of the report script and how to incorporate the script into the screen JPL module. (Refer to Chapter 5 for a complete description of the script statements and how to use them to direct report processing and composition.)

Once the report format screen is completed, it must be compiled before it can be used to generate a report. This step is described in Section 4.4.

If C (or other supported language) functions are invoked by the report script, they must be installed on the control function list or the prototyped function list and linked with the ReportWriter executable. This procedure is summarized in Section 4.5.

4.1 REPORT AREA LAYOUT

The report format screen serves as a template for the finished report. All report areas can be included on this screen. Lay out the fields and display text just as you would if you were designing a screen for a JAM application.

You may also choose to store some of the report areas in separate report format screens, especially those areas that are used in several different reports. The information in this section applies to all report format screens, whether they contain entire reports or only an

area or two. (Section 4.3.2 explains how to compose a report from multiple report format screens using the `<<include screen>>` compiler directive.)

4.1.1

Name Tags

Since ReportWriter allows multiple report areas on a single screen, you must include a name tag at the end of each line to identify which area the line belongs to. Multiple area names can be entered in the tag. Thus, if the same line appears in more than one report area, you can label it with the names of all relevant areas.

The syntax for name tags is

<<areaname [areaname...]

Put the name tag to the right of all fields and display text on the line. It can immediately follow the last field or display area on the line:

```
SAMPLE REPORT                    3/11/92 << exphead
<< exphead exshead-dr exshead-cr
STATE   CITY           DEBIT << exshead-dr
STATE   CITY                  CREDIT << exshead-cr
------------------------------------ << exshead-dr exshead-cr
<< exshead-cr exshead-dr
_____   ____________   _____   ______<<exdetail
<<expfoot
                Page ___ << expfoot
      Acme, Inc.      Wallahasoo, TN << expfoot
```

Or you may prefer to line up all the name tags for improved readability:

```
SAMPLE REPORT                    3/11/92 << exphead
                                         << exphead exshead-dr exshead-cr
STATE   CITY           DEBIT             << exshead-dr
STATE   CITY                     CREDIT  << exshead-cr
-----   -------------  -----     ------  << exshead-dr exshead-cr
                                         << exshead-cr exshead-dr
_____   ____________   _____     ______  << exdetail
                                         << expfoot
              Page ___                   << expfoot
      Acme, Inc.      Wallahasoo, TN     << expfoot
```

ReportWriter's use of << as the name tag delimiter does not preclude the report designer from using this symbol in a report display area. The rightmost appearance of this symbol on the line is interpreted as the start of the name tag; all characters to the left of it are part of the screen line.

NOTE: Since the underbar is **JAM**'s default draw field symbol, you should avoid this character in area names. Unless you select a different draw field symbol, the **JAM** screen editor will interpret the underbar as a one–character field instead of part of the area name tag. In this manual, hyphens are used in multi–part area names.

4.1.2

Organizing the Report Format Screen

ReportWriter imposes no restrictions as to the order in which report areas are placed on the consolidated screen.

You may find it helpful, for example, to follow a break header area with its corresponding detail area so that you can more easily line up the columns with their titles, or you may prefer to keep break headers and footers at the same level adjacent to each other. If you are using a title page for the report, you can put it first, last, or anywhere in between.

In short, the order of report areas on the format screen is of no consequence. Their order in the final report is controlled by the report script, not by the order in which they appear on the format screen.

Lines belonging to the same area do not have to be contiguous on the screen. Normally, you would want to keep an area intact so that you can see its layout more clearly. The exception would be if you use multiple tags to avoid repeating identical lines.

While the lines for a given area do not have to be kept together on the format screen, they must, however, appear in the proper order. Suppose, for example you have a report with the following break headers:

Break header 1:

```
COLUMN-A              COLUMN-B
```

Break header 2:

```
COLUMN-C    COLUMN-D        COLUMN-E
```

You could draw these report areas in any of several different ways, some taking advantage of the fact that the horizontal rules repeat in both headers, and some that do not use this capability:

Method 1 draws each area individually:

```
_____________________________________<<bhead1
                                     <<bhead1
COLUMN-A          COLUMN-B           <<bhead1
_____________________________________<<bhead1
_____________________________________<<bhead2
                                     <<bhead2
COLUMN-C    COLUMN-D       COLUMN-E  <<bhead2
_____________________________________<<bhead2
```

Method 2 takes advantage of some of the duplication:

```
_____________________________________<<bhead1  bhead2
                                     <<bhead1  bhead2
COLUMN-A          COLUMN-B           <<bhead1
_____________________________________<<bhead1
COLUMN-C    COLUMN-D       COLUMN-E  <<bhead2
_____________________________________<<bhead2
```

Method 3 takes maximum advantage of the duplication:

```
_____________________________________<<bhead1  bhead2
                                     <<bhead1  bhead2
COLUMN-A          COLUMN-B           <<bhead1
COLUMN-C    COLUMN-D       COLUMN-E  <<bhead2
_____________________________________<<bhead1  bhead2
```

Note that in each of the preceding screen layouts, the correct ordering of lines in the break headers is preserved. Do *not* draw these areas, for example, as shown here:

```
_____________________________________<<bhead1  bhead2
                                     <<bhead1  bhead2
COLUMN-A          COLUMN-B           <<bhead1
_____________________________________<<bhead1  bhead2
COLUMN-C    COLUMN-D       COLUMN-E  <<bhead2
```

The above example formats break header 1 correctly but would produce the following for break header 2:

```
_____________________________________

_____________________________________
COLUMN-C    COLUMN-D       COLUMN-E
```

4.1.3

Field Names

In JAM, all field names on a screen must be unique. When designing a report, however, you may want the same field to appear in more than one report area. ReportWriter provides a field–naming syntax that allows you to equate fields on the report format screen while observing JAM's restriction against duplicate field names:

base.extension

One field, either on the screen or in the LDB, must be named ***base*** (with no extension).

Use the same base name for all other fields identical to this one, and distinguish them with unique extensions. For example, suppose the field `cust_name` appears in the detail area, in the break footer, and in a summary area. One must be named `cust_name`; its clones might be named `cust_name.1` and `cust_name.2`.

You can choose whatever extensions you prefer. In the example above, 1 and 2 were used for simplicity. You may prefer to have the extensions identify the report areas, as in `cust_name.dtl` and `cust_name.bfoot` (assuming that you are using `cust_name` with no extension in the summary area). As long as the base names are the same, ReportWriter will recognize these as identical fields.

The extension delimiter must be a period.

When using these fields as variables in the report script and in JPL code, refer only to the base name. In the above example, you might have the customer name designated as a break field:

```
# break field   = cust_name
```

Similarly, JPL code should reference the base name only:

```
cat cust_name first " " last
```

Make sure that all fields with the same base name are identical in length and format. ReportWriter operates only on the base field (with no extension). Whenever one of its clones is output, ReportWriter copies the current contents of the base field into the clone. If the clones have different characteristics than the base field, unexpected results might occur on output.

The `base.extension` syntax is only required for fields on the report format screen that must be equated. Fields that appear in one area only can be named as you would any other JAM screen field.

4.1.4

Fields That Do Not Appear in Output Areas

In some cases, you will need to define fields that can be used as screen–level variables even though these fields are not intended to appear in any report area.

For example, you can define breaks based on values computed in JPL procedures, but JPL variables do not have the scope required for break fields. Screen fields, however, do have the correct scope, so a simple way to handle this situation is to create screen fields for these variables. (Refer to Section 5.2 for a more complete discussion of break fields and break processing.)

If your report requires such fields but they are not needed in any report output areas, place these fields on any line of the report format screen that does not have an area name tag. Such fields will be available to all JPL procedures and ReportWriter processing in the same way as fields in the tagged areas. Make sure that you name the fields and that these names match those of the corresponding break fields and JPL variables.

A field that appears on the report format screen but is not on a line containing an area name tag is a *non–output field.*

Non–output fields can also be used as target variables for a database fetch, temporary variables, date/time fields, and so forth. Since they are screen fields, their scope makes them useable in the report script or wherever needed in the JPL procedures associated with the screen.

Use of non–output fields can minimize or eliminate the need to put report variables in the LDB.

4.2

THE REPORT SCRIPT

Enter the report script into the screen JPL module of the report format screen. Each line of the script must begin with the JPL comment indicator, #. The script can appear anywhere within the JPL module and is delimited by the lines

```
# << begin report >>
```

and

```
# << end report >>
```

A simple report script might look like:

```
# << begin report >>
# init    jpl = dbinit
#         lines  = 24 /* length of multi-part invoice form */
#
# page header  jpl = initcount
```

```
#              area    = phead
#
# detail    query   = "select * from invoices \
#                       order by acct_id"
#           area    = inv-detail
#           jpl = detail_jpl
#
# break     field   = acct_id
#           footer  area = bfoot
# << end report >>
```

4.2.1

Structure of the Script Language

The report scripting language comprises the following six statements:

`break`
: defines a "break" field and the action to occur when the value of this field changes

`clear`
: cancels all previous page and/or break specifications

`detail`
: specifies the SQL statement used to fetch data for a row–level report section; also specifies the report area(s) used to format the fetched data in the report and any routines to be called for additional processing

`init`
: performs report initialization

`insert`
: specifies report area(s) to be inserted, a subreport to be invoked, or a JPL or C routine to be executed on a one–time basis

`page`
: specifies page headers and footers and any associated processing

In the discussion that follows, statements are described as having associated clauses, subclauses, and keywords.

The broad use of the term *keyword* refers to any string that has a specific meaning in the report script—the six statement names, and all statement modifiers. In the narrower sense, keyword refers to statement modifiers that stand alone without additional modifiers or arguments. For the sake of clarity, the term *reserved word* is sometimes used in place of the broader meaning of keyword.

The term *clause* is used to refer to a modifying keyword that requires an argument of some sort, such as the name of a report area or a JPL module, or that can take additional

modifiers. The term *subclause* refers to any clause that is incorporated within another clause.

Consider the following `break` statement:

```
# break field      =   authname
#
#          footer  call    =   subtots
#                  area    =   authfoot
#                  call    =   zerotots
#
#          header  area    =   newauth
#                  showattop
#
#          norepeat
```

This statement consists of `field`, `footer`, and `header` clauses and the keyword `norepeat`. The `footer` clause consists of two `call` subclauses and an `area` subclause. The `header` clause contains an `area` subclause and the keyword `showattop`.

The six report script statements are described more thoroughly in Chapter 5 of this manual. The complete syntax for each is provided in Chapter 11, the script statement reference. The format of the report script is described below.

4.2.2

Format of the Report Script

This section describes the formatting requirements of the report script within the screen JPL module.

JPL Comment Indicator

Since the report script is incorporated as a comment in the JPL module, each line must begin with a pound sign (#), the JPL comment indicator.

Delimiting the Report Script

The script can appear anywhere within the JPL module and is delimited by the lines

```
# << begin report [= name] >>
```

and

```
# << end report >>
```

Each subreport in the script, as well as the primary report, must be delimited by these compiler directives. The report name is optional for the first report in the script. All subsequent reports, however, must be identified by a name.

Refer to Sections 4.3.1 and 6.2 for a more detailed explanation of these compiler directives.

White Space

At least one space, newline, or tab character must separate each keyword (statement or clause name) from the text that precedes it.

For example,

```
# page header area = phead
#      footer jpl  = pfoot
#             area = pfoot
```

is equivalent to

```
# page header area=phead footer jpl=pfoot area=pfoot
```

While each of the two examples above is acceptable input to the script compiler, the style of the first is recommended for its readability.

White space is not significant in report script delimiters. The following are all interpreted correctly by the script compiler:

```
#<<begin report>>
#     <<begin     report     >>
# <<   begin report >>
```

Note, however, that the words `begin` and `report` must be separated by at least one space. Similarly, in the ending delimiter, `end` and `report` must be separated by at least one space.

Comments

Comments can be included in the report script. Any text appearing between `/*` and `*/` is interpreted as a comment. Remember that lines containing script comments must begin with the JPL comment indicator, #, as do all other lines in the script.

```
# /* This entire line is a comment. */
# init columns = 80/*This comment follows a statement.*/
# /* This is a
#    multi-line
#    comment */
```

Blank lines

Blank lines can appear anywhere in the script.

For example,

```
# init columns = 80
#
# page header area = phead
#      footer area = pfoot
```

is equivalent to

```
# init columns = 80
# page header area = phead
#      footer area = pfoot
```

or

```
# init columns = 80
# page header area
#
#                  = phead
#      footer area = pfoot
```

In the first of the examples above, the blank line between statements is used to improve readability of the script.

It is recommended that blank lines begin with the JPL comment indicator, even though this is not a requirement. (The report script compiler excludes uncommented blank lines in assigning the line numbers which appear in error messages; therefore, beginning every line with a comment indicator will probably make it easier for you to locate any line referenced in an error message.) Throughout this manual, all blank lines begin with the JPL comment indicator.

Use of Quotation Marks

Non–numeric (string) arguments to script keywords and compiler directives are enclosed in quotation marks. The quotation marks can be omitted if the argument string consists only of letters, digits, hyphens, periods, or the underbar character.

If the argument contains blanks or special characters, or if the argument is the null string, the quotation marks are required.

Quotation marks are required in the following examples:

```
# call  = "dototals credits debits"
# area  = "h/foot"
# query = "SELECT * FROM accts"
# jpl   = "getname last first"
# area  = ""
```

Quotation marks are optional in these examples:

```
# call  = initval
# area  = detail2
# jpl   = cleartots
# <<include screen = screen.jam>>
```

Continuation Character

If a quoted string occupies more than one line, a backslash (\) is used as the continuation character.

The backslash must be the last character on the line. The SQL statement that appears in the `query` clause of a `detail` statement frequently requires more than a single line:

```
# detail area = datascrn
#        query ="SELECT emp.grade, emp.first, emp.last,\
#                  acc.sal \
#                FROM emp, acc WHERE emp.ss=acc.ss \
#                ORDER BY emp.grade, emp.last, \
#                  emp.first"
```

Note that the JPL comment indicator is required at the beginning of each continued line.

Case Sensitivity

Keywords in report script statements and clauses are not case–sensitive.

Remember, though, that this applies only to script file keywords and not to file names (if your operating system is case–sensitive) or to the text of SQL statements.

The following two script statements are equivalent:

```
# INIT COLUMNS = 80
# init columns = 80
```

Similarly, the script delimiters are not case–sensitive:

```
# << END REPORT >>
# << End Report >>
# << end report >>
```

Any other permutation of upper– and lower–case is also accepted by the script compiler.

SQL Statements

The great latitude available in formatting and capitalization within the report script does not apply to the SQL statement that appears in the `query` clause. The SQL statement must conform to the formatting and capitalization requirements of the database in use.

4.3 COMPILER DIRECTIVES

4.3.1 Script Delimiters

```
<<begin report>>
<<end report>>
```

Each report or subreport in a report format screen must be delimited by the `<<begin report>>` and `<<end report>>` compiler directives to indicate the beginning and end of each script.

The syntax of the `<<begin report>>` compiler directive allows you to specify a name for each report:

```
<< begin report [= name] >>
```

where:

name is the name by which the report or subreport is invoked; it is required for any report that is called as a subreport

The names of all reports defined and/or included in a report format screen must be unique in the first 21 characters.

If the first report defined in a report format screen does not have the ***name*** argument, it is known by the name of the screen and can be invoked only as a primary report. If you want to invoke it as a subreport, you must specify the ***name*** argument in the `<<begin report>>` compiler directive. This argument is optional only for the first report script associated with a screen; it is required for all subsequent scripts. Only the first script in a report format screen can be invoked as a primary report.

Each `<<begin report>>` must be paired with an `<<end report>>` compiler directive following the corresponding script. Nesting of report scripts is not permitted.

Refer to Section 6.2 for further information on subreports.

4.3.2 Using Multiple Report Format Screens

```
<<include screen>>
```

If several of your reports share a common layout for a particular area, a standard page header, for example, or a common trailer page, you can create the layout once as a sepa-

rate JAM report format screen and direct the compiler to include it in each relevant report.

Similarly, you may store some or all subreports in separate report format screens. Each report that invokes these subreports must direct the compiler to include the relevant report format screen(s).

To include another report format screen in your report, add the following compiler directive to the script:

```
# << include screen = scrnname >>
```

where ***scrnname*** is the name of the report format screen containing the area(s), script(s), and/or JPL you want to add to the main report.

This compiler directive must appear after the

```
# << begin report >>
```

delimiter but before any script statements.

In the following example, `headfoot.jam` contains layouts for the page headers and footers; these areas are tagged `phead` and `pfoot`, respectively. A second screen, `title.jam`, is also included; it contains an area named `titlepg`.

```
# << begin report >>
# << include screen = headfoot.jam >>
# << include screen = title.jam >>
#
# init      jpl     = initjpl
#           area    = titlepg
#
# page  header  area    = phead
# page  footer  area    = pfoot
#
.
.
.
# << end report >>
```

The `<<inlcude screen>>` compiler directive imports the area layouts, report scripts, and screen JPL from the specified report format screen.

Scripts imported from included screens are appended to the end of the primary report script. They are added in the same order as the `<<include screen>>` directives are processed.

Similarly, screen JPL code is appended to the primary report format screen's JPL module.

The following information is omitted from an included screen when it is imported into the primary report format screen:

- border and background color
- screen edits
- control strings

Any number of additional screens can be included, as long as the total number of lines in the merged screen does not exceed 254. Once the merged screen has reached the limit of 254 lines, no further display text or fields can be imported.

The screen(s) to be included can contain any number of report areas, as long as each is correctly tagged.

If a field name on an included screen matches a field name on the main screen, the script compiler adds an extension to the field on the included screen and treats it as a clone of the similarly–named field on the main screen. The extension chosen is the name of the included JAM screen.

Refer to Section 4.1.3 for more information on naming identical fields in separate report areas.

If an area name tag on an included screen matches one on the main screen, that area of the included screen is appended to the corresponding area of the main screen. If the same area name tag appears on multiple included screens, they are appended to the main screen in the order of the `<<include screen>>` compiler directives.

Nested Includes

If your report script imports another report format screen that also contains `<<include screen>>` compiler directives, you will want to be aware of the order in which the screens are appended to the primary report format screen. This is an issue only when multiple imported screens contain areas, reports, or procedures bearing identical names.

Each `<<include screen>>` compiler directive in the primary report format screen is processed to its full depth before the next screen is imported.

Suppose, for example, the primary report format screen contains the following `<<include screen>>` compiler directives:

```
<< include screen = x >>
<< include screen = y >>
```

and screen x contains

```
<< include screen = z >>
```

The resulting merged screen would begin with the entire primary report format screen, followed by screen x, screen z, and, finally, screen y.

As mentioned above, if an area name tag on an included screen matches one on the primary screen, that area of the included screen is appended to the corresponding area of the main screen. If the same area name tag appears on multiple included screens, they are appended to the primary screen in the order the `<<include screen>>` compiler directives are processed.

To continue the example, suppose the primary report format screen, screen `y`, and screen `z` all contain areas with the tag `a`. The resulting area `a` on the merged screen would consist of all the `a` lines from the primary screen, followed by those from screen `z` and then from screen `y`.

If you are developing reports that import screens containing `<<include screen>>` compiler directives, you should keep in mind the order in which these directives are processed and the screens appended to the primary screen.

4.4
COMPILING THE REPORT

When you have completed the report format screen—laying out the report areas, entering the report JPL into the screen JPL module, and entering the report script as a comment in the screen JPL module—you must compile the report with the `rprt2bin` utility.

On the command line, type

```
rprt2bin [-e ext] rprtname
```

where ***rprtname*** is the name of the report format screen to be compiled. If ***rprtname*** does not include an extension, the default extension specified in the environment variable `SMFEXTENSION` is assumed.

The output of `rprt2bin` is a binary file named ***rprtname.ext***. If the `-e` option is omitted, the resulting file is named ***rprtname***`.bin`.

The report binary file will be used as input to `rwrun`, the report generation utility.

The report binary file cannot be edited. If you need to change the report in any way after it has been compiled, you must make the changes to the source file for the report format screen and recompile.

The report format screen to be compiled must be in ReportWriter release 5 format. Use the `rw4to5` utility, described in Chapter 13, to convert your release 4 reports to release 5 format before attempting to compile them.

4.5 INSTALLING CALLED FUNCTIONS

Any function invoked by a `call` clause in the report script must first be installed on either the control function list or the prototyped function list and linked into the ReportWriter executable.

The steps below outline the general procedure for installing and linking your compiled C (or other supported language) functions into ReportWriter. The details of this process are operating system dependent. Refer to the files provided in your ReportWriter distribution for more specifics.

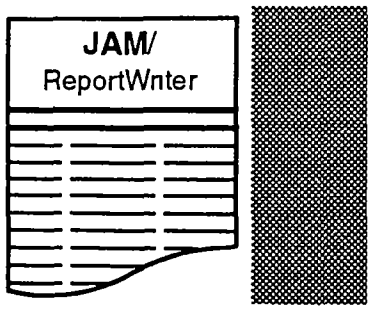
1. Install the functions into the control function list or the prototyped function list of `funclist.c`. Instructions for this step are provided in the **JAM** *Programmer's Guide* and in the `funclist.c` file, which is part of the **JAM** distribution.
2. Compile `funclist.c`.
3. In `rwmain.c`, make sure that the line

   ```
   sm_do_uinstalls ()
   ```

   is not commented out.
4. In the supplied makefile, append the names of the object files for your functions onto the line that begins

   ```
   USERMODS = funclist.o
   ```

   This step is documented in the makefile. (The exact name of the makefile depends upon your operating system.)
5. Run the makefile to create the desired executables. This step recompiles `rwmain.c` and relinks `rwrun`, `jamrw`, and/or `jxrw`, depending upon your other modifications to the file. (Refer to instructions in the makefile for specifying which executables to create.)



Chapter 5
Using the Script Statements

This chapter explains how to use the script statements to produce a report with the required content and formatting.

The first three statements described, `detail`, `break`, and `insert`, deal primarily with generating report content—capturing, processing, and formatting report data.

The remaining three report script statements, `init`, `page`, and `clear`, deal primarily, but not exclusively, with page formatting: dimensions, margins, page headers, and page footers. Each of these statements can also interact with report contents in such ways as

- using `init` to invoke a routine that logs on to the database,
- inserting the current value of a report variable into a header specified in the `page` statement, or
- using `clear` to terminate prior break specifications.

The six report script statements are described functionally in the sections that follow. For a concise syntactical description of these statements, refer to Chapter 11.

5.1
SPECIFYING THE DATABASE QUERY: `detail`

The `detail` statement is used to specify the database query that fetches data for a row–level or summary report.

NOTE: If a developer–written row–supply function is installed, the detail query is passed to this function rather than to JAM/DB*i*. Refer to Section 8.3 for information on developer–written row–supply functions.

Each `detail` statement must include a single `query` clause or `cursor` clause. In addition, this statement can contain any number of `area` , `report`, `jpl`, or `call` clauses. The keywords `newpage` and `split` can also be included as necessary to control formatting of detail report areas. The keyword `breakcheck` may be added to specify where break processing should occur. In addition, the `area` clause can be modified by the `shrink` keyword.

A typical `detail` statement might look like

```
# detail    query    =    "SELECT * FROM accts \
#                          ORDER BY state, acctid"
#           breakcheck
#           jpl      =    dtlcalc
#           area     =    acctdtl shrink
#           jpl =    "acctproc cumtots"
#           newpage
```

5.1.1

Detail–Level Processing

When a `detail` statement is encountered in the report script, ReportWriter processes the SQL statement in the `query` clause or executes the named cursor specified in the `cursor` clause and, upon each fetch from the database, executes the `area`, `report`, `jpl`, and `call` clauses. ReportWriter continues to cycle through the detail statement in this manner until no more rows are fetched from the database.

The `area`, `report`, `jpl`, and `call` clauses and the `breakcheck` keyword are executed in the order they are encountered within the `detail` statement. Thus, any procedures that compute values required for the output areas must be invoked with `jpl` or `call` clauses placed before the applicable `area` clauses. Similarly, if any procedures modify data in ways that you do not want reflected in the current cycle's output, place the corresponding `jpl` or `call` clauses after the corresponding `area` clause.

If break processing is enabled for this section of the report, the order in which rows are fetched must be consistent with the hierarchy of break fields. The SQL statement in the `query` clause (or the named cursor) must include an `ORDER BY` clause specifying the same fields as the break fields and in the same order. Refer to Section 5.2 for more information on break processing.

In the event that no rows match the query, no detail processing or output is performed. ReportWriter issues a warning message in this case. (Warning messages can be suppressed by using the `-i` option when ReportWriter is invoked; refer to Sections 9.1 and 9.4 for more information on this option.)

5.1.2

detail Clauses and Keywords

query

The query clause consists of a SQL statement that fetches the desired rows from the database. Note that the SQL statement must be a SELECT statement[1] and must conform to the syntax of the database in use. The SQL statement must be enclosed in quotation marks.

If a developer–written row–supply function is used in place of JAM/DB*i* to provide input to ReportWriter, the query must conform to the syntax expected by the input function.

This clause can appear anywhere within the detail statement. Typically, it is either the first or last clause, but this is not required, since its position does not affect the order of processing.

As each row is fetched from the database, ReportWriter must output an area, perform some processing, or both. The area, jpl, and call clauses specify the appropriate actions.

cursor

A named cursor can be invoked as an alternative to the query clause in the detail statement.

You must declare the cursor in a JPL or C routine that will be executed before the corresponding detail statement is encountered. Execute the cursor by using a cursor clause instead of a query in the detail statement. The syntax for the cursor clause is

cursor = ***invocation_string***

where:

invocation_string
: consists of the name of the cursor followed by the applicable arguments, if any. A space separates the cursor name and the arguments, if any. The arguments themselves should be formatted according to your JAM/DB*i* specifications for the dbms statement EXECUTE USING. The arguments are colon expanded and then passed as a string to

1. If you are using Sybase, the query clause can also be used to invoke a stored procedure. For example:
```
query = "exec procname :arguments"
```

```
dbms WITH CURSOR cursor EXECUTE USING argument_string
```

if ReportWriter is linked with JAM/DB*i* release 5, or to

```
dbms EXECUTE cursor USING argument_string
```

if ReportWriter is linked with JAM/DB*i* release 4.

The invocation string must be enclosed in quotation marks if it contains embedded spaces or other special characters.

Each `detail` statement must contain one `query` clause or `cursor` clause, but not both.

Refer to Section 6.7 for more information on using named cursors in the `detail` statement.

area

The `area` clause specifies a report area to be output for each fetch from the database. This clause is optional; multiple `area` clauses are permitted in a `detail` statement.

Use the `shrink` keyword to remove excess blank fields from the specified report area on output. This feature is particularly useful for report areas containing arrays that may not be fully populated. If `shrink` is specified, the trailing unused elements are removed and the report area is condensed accordingly. If the entire field, which could have one or more onscreen elements, is blank, the entire field is removed. This keyword removes only blank lines, not columns, thus shrinking the area vertically, but not horizontally. Place the `shrink` keyword after the area name:

```
# detail    area    =    empdtl shrink
```

Refer to Section 6.5.1 for more information on sizing areas dynamically.

If the `breakcheck` keyword does not appear in the `detail` statement, placement of the first `area` or `report` clause determines the point at which break processing is performed. By default, break checking and processing occur immediately before the first `area` or `report` clause, or, if neither of these clauses is present, after all `jpl` and `call` clauses in the `detail` statement have been executed. Refer to Section 5.2 for a more complete explanation of break checking and processing.

If you are creating a summary report that displays only totals and subtotals rather than the values for each row in the database, you can omit the `area` clause. (Remember, however, that with no `area` or `report` clause present, you will probably need to use the `breakcheck` keyword to ensure that break checking and processing occur at the correct point in the execution flow of the `detail` statement. The `breakcheck` keyword is described below.)

report

The report clause specifies a subreport to be invoked for each fetch from the database. This clause is optional; multiple report clauses are permitted.

If the breakcheck keyword does not appear in the detail statement, placement of the first report or area clause determines the point at which break processing is performed. By default, break checking and processing occur immediately before the first area or report clause, or, if neither of these clauses is present, after all jpl and call clauses in the detail statement have been executed. Refer to Section 5.2 for a more complete explanation of break checking and processing.

Refer to Section 6.2 for a complete discussion of subreports.

jpl

The jpl clause specifies a JPL procedure to be executed for each fetch from the database. This clause is optional; multiple jpl clauses are permitted.

call

The call clause specifies a C (or other supported language) routine to be executed for each fetch from the database. This clause is optional; multiple call clauses are permitted.

newpage

The newpage keyword forces a page break after output for each cycle through the detail statement is completed. This keyword is ignored if no report area or subreport with output is specified in the detail statement.

Use the newpage keyword to ensure that each row's output begins on a new page.

split

By default, ReportWriter will not unnecessarily split a multi-line area between pages. If there is insufficient room on the current page to accommodate the entire report area, ReportWriter will force a page break and begin it on a new page.

The split keyword overrides this pagination rule, permitting the area to begin part way down on a page, even if that means it will be split across pages.

The split keyword must be placed with the area clause to which it applies.

breakcheck

This keyword indicates where in the sequence of `area`, `subreport`, `jpl`, and `call` clauses break processing should occur. For example, in the following detail statement, break processing takes place after the JPL procedure `prebreak` is executed and before the procedure `postbreak` is invoked:

```
# detail    query    = "SELECT ..."
#           jpl      =   prebreak
#           breakcheck
#           jpl      =   postbreak
#           area     =   dtlarea
```

If the `breakcheck` keyword is not specified, break processing takes place immediately before the first area output or subreport invocation (or after all `jpl` and `call` clauses, if no `area` or `report` clause is present). In the above example, if the `breakcheck` keyword were omitted, break processing would take place after execution of the JPL routine `postbreak`.

The `breakcheck` keyword is also useful for retaining the pre-break value of a break footer field computed from a fetched field.

Suppose, for example, that the procedure `set_b` computes the value of field `b` from that of fetched field `a`. Suppose, too, that `b` is a field in the break footer area, so it must show, when the break footer is output, the value computed from the previous fetch rather than that corresponding to the current fetch. To ensure that the correct value will be displayed, simply delay computing its value from the current fetch until after the break footer has been output.

Consider the following two code fragments:

```
# /* code fragment (1): */
#
#       break   field = a
#               footer area = bfoot
#
#       detail  query = "SELECT a FROM t ORDER BY a"
#               jpl   = set_b
#               area  = dtl-area


# /* code fragment (2): */
#
#       break   field = a
#               footer area = bfoot
#
#       detail  query = "SELECT a FROM t ORDER BY a"
#               breakcheck
#               jpl   = set_b
#               area  = dtl-area
```

In fragment (1) the value of b is computed before break processing. If break checking determines that field a has broken (and this row, therefore, begins a new break group), the prior value of b will have already been lost, and the new value will appear in the footer for the previous break group.

In fragment (2), the computation of variable b is delayed until after break checking and processing. Therefore, when the break footer for the previous break group is output, the value shown for b will be that computed from the correct value of the break field. After break processing, b is recomputed so that it will show the correct value based on the current value of field a.

Refer to Section 5.2.3 for additional information on retaining the pre–break values of break footer fields.

The breakcheck keyword is effective only if it appears ahead of the first area or report clause.

Refer to Section 5.2 for more information on break processing.

5.2
DEFINING BREAK FIELDS AND PROCESSING: break

The break statement is used to define a *break field* and to specify the associated processing and/or output that should occur when there is a change in the value of this field. The break field must be either a field on the report format screen or an LDB variable.

A JPL variable cannot be used as a break field. You can avoid having to use the LDB by creating a non–output field of the appropriate type on the report format screen. Refer to Section 4.1.4 for more information on entering non–output fields onto the screen.

Each break statement must specify a break field and, optionally, a break header, break footer, or both. In addition, the keywords norepeat, norepeatattop, and newpage can be included, if applicable, to control formatting. Output areas in the break header clause can be modified by the nodupl, split, shrink, and showattop keywords; output areas in the footer clause can be modified by the nodupl, shrink, and split keywords. The break footer can also be modified by the keyword noorphanbreak.

A typical break statement might look like:

```
# break     field   =   authname
#
#           footer  call    =   "subtots authtot salestot"
#                   area    =   authfoot
#                   call    =   zerotots
#
#           header  area    =   newauth showattop
#
#           norepeat
```

5.2.1

Hierarchy of Break Fields

Multiple `break` statements define a hierarchy of breaks. The first `break` statement identifies the highest–level break field. Subsequent `break` statements define breaks at increasingly lower levels. Whenever a break occurs, all lower–level breaks are automatically forced to occur at the same time.

The following example defines breaks by genus, species, and subspecies:

```
# break     field   =   genus
#           header  area    = head
#           footer  area    = gfoot
#
# break     field   =   species
#           footer  area    = sfoot
#                   jpl     = sfoot
#
# break     field   =   subspecies
#           footer  area    = ssfoot
#                   jpl     = ssfoot
```

Whenever the value of a break field is found to have changed, break processing is initiated, not only for that field, but also for all lower–level break fields. Thus, you can cause a given set of break field actions to occur if any one of several specified fields is changed.

To do so, define each such field as a break field, and associate the desired action with only the lowest of these breaks. If any one or more of the specified fields change value, the indicated actions are invoked.

In the following example, the area `name-header` is output whenever any part of the name changes:

```
# break     field   = last_name
# break     field   = first_name
# break     field   = middle_initial
#           header  area = name-header
```

5.2.2

Break Field Processing

The `detail` statement causes all rows matching its query to be fetched from the database and processed. Before the first fetch is processed, break header procedures and areas are executed and output, from the highest level break to lowest. (If no rows match the detail query, no break processing or output is performed.)

As subsequent rows are processed, each break field is checked to determine if its value has changed since the previous row was processed. If no break is detected (that is, the values of the break fields have not changed), the new row is processed as indicated in the `detail` statement.

Break checking is performed immediately before the point in the `detail` statement where break processing would occur, if needed; that is, where the `breakcheck` keyword is encountered, or, if `breakcheck` is not present, immediately before the first `area` or `report` clause, or, if neither is present, after all `jpl` and `call` clauses have been executed.

If the value of a break field is found to have changed, break processing is initiated for that field and all lower–level break fields, in the following order:

1. The procedures and areas for break footers are executed and output, beginning with the lowest level break field.
2. Header procedures and areas for the broken fields are processed similarly, but in the opposite order—highest level to lowest.
3. Processing of the current database fetch resumes.

Break specifications remain in effect until a `clear` or `clear breakspecs` statement is encountered in the report script.

`break` statements can appear anywhere before or after the `detail` statements to which they apply, as long as there is no `clear` or `clear breakspecs` statement between the `break` and `detail` statements.

Similarly, `break` statements that define a hierarchy need not immediately follow each other. (Although, for the sake of clarity, you should place the `break` statements together in the script so that the hierarchy is obvious.) Once a `clear` or `clear breakspecs` statement is encountered, however, additional `break` statements define a new break hierarchy, rather than adding to the previous one.

Where break processing is enabled, the order in which rows are fetched from the database must be consistent with the hierarchy of break fields. The SQL statement in the `detail query` clause must include an `ORDER BY` clause specifying the same fields as the break fields and in the same order. Refer to Section 5.1 for more information on detail processing.

The order of clauses and keywords within the `break` statement is not significant. The order of `area`, `report`, `jpl`, and `call` subclauses within the `footer` and `header` clauses, however, is significant and determines the order in which processing and area output occur.

5.2.3

Retaining Pre–Break Values

Any break field that appears in a break footer area will show its pre–break value when the footer is output. Non–break fields whose values are fetched by the detail query will normally contain, at break–footer time, the values associated with the detail row that will begin the following break group. To force these fields to show their pre–break values in break footers, define them as lowest–level break fields with no actions, such as:

```
# break field   =   name
# break field   =   address
```

Any field computed from a fetched field will also show its post–break value when it appears in a break footer unless

- it is defined as a lowest level break field (as described above), or
- the detail processing that computes its value is invoked after break processing takes place. (Refer to the `breakcheck` keyword description for an example of this sequence.)

5.2.4

Computed Breaks

At times, you will need to define breaks on variables that are not fetched from the database.

For example, in a long row–level report, you may want to insert a blank line every few rows to make the report easier to read. Or, you may want your break field to be a value derived from data fetched from the database, such as ordering rows by date and then grouping them by quarter of the year. Alternatively, you may need to output an additional report area each time a row of data meets some specified criterion, as in an employee listing where you want to show the territory information for each employee whose department is "sales."

In each of these cases, the break field is not a column in the database but, rather, a variable whose value is generated in a JPL or C routine. The variable may be simply a flag whose

value is changed for the sole purpose of triggering a break, as in the first and third examples in the preceding paragraph, or it might bear a more direct relationship to the data in the row, as in the second example.

As mentioned earlier, break fields must be either screen fields or LDB variables. Since the break field in a computed break is not likely to appear in a report output area, remember to define a non–output field on the report format screen so that this variable can be used as a break field. This procedure is described in Section 4.1.4.

The following example is similar to the second scenario, above, in that the value of the break field is derived from the fetched data. Suppose you are reporting questionnaire results and you want to group the responses by age range: under 18, 18 – 25, 25 – 40, and so forth. (Assume that each respondent's age is included in the database.)

To implement this:

- On the report format screen, create a non–output field to use as the break field. You may want to call it `age_range`. The procedure for creating non–output fields is described in Section 4.1.4.
- Write a JPL procedure that computes `age_range` for each fetched row.
- Define a break on the `age_range` field.
- Order the fetched rows by age.

The `break` and `detail` statements to implement this break might look like:

```
# break    field  = age_range
#          footer jpl    = fill_in_range
#                 area   = age-group-summary
#                 jpl    = reset_totals
#          header area   = column-titles
#
# detail   query = "SELECT * FROM responses\
#                    ORDER BY age"
#          jpl    = figure_age_range
#          area   = detail-area
#          jpl    = do_running_totals
```

The JPL procedure `figure_age_range`, invoked each time a row is fetched, would include the following:

```
proc figure_age_range

# determine which age range the respondent for this
# row belongs in

# the ranges are:  1: under 18
#                  2: 18 - 25
#                  3: 26 - 40
```

```
#                   4: 41 - 60
#                   5: 61 and over

# the variable age_range is a non-output field
# on the report format screen

if age >= 61
    cat age_range "5"
else if age >= 41
    cat age_range "4"
else if age >= 26
    cat age_range "3"
else if age >= 18
    cat age_range "2"
else
    cat age_range "1"
```

Since `figure_age_range` computes the break field value for the current row, it must be invoked before break checking takes place. The `detail` statement (shown on the preceding page) implements this required order of processing.

You will probably also want some text on the break footer area to identify which age range the responses belong to. Write a JPL procedure, called `fill_in_range` in this example, to supply the applicable text for a field on the `age-group-summary` area (the break footer). Your procedure will determine the appropriate text from the value of the break field, `age_range`.

Note that ReportWriter saves the prior value of any variable defined as a break field; it uses the old value, rather than the new one, in all break footer processing. Even though the value of `age_range` will have changed by the time `fill_in_range` is invoked, it will access the correct (i.e., the one that pertains to the just–printed group) value of `age_range`.

5.2.5
`break` Clauses and Keywords

`field`

The `field` clause specifies the break field. This field must appear on the report format screen or be in the LDB. Typical break fields include columns fetched from the database and values computed in JPL or C routines.

If the break field is a computed value that is not propagated to a field in a report area, you will have to create a non–output field on the report format screen or else place the variable in the LDB.

The field name can, optionally, be modified by a substring specification, giving the character position for starting the comparison and the number of characters to compare. Suppose, for example, a company uses a part numbering scheme in which characters 3 through 7 identify the manufacturer. To produce an inventory report in which the manufacturer is the break group, specify the relevant substring of the part number as the break field:

```
# break    field  = partno (3,5)
```

Array elements can also be used as break fields. For example:

```
# break field = city[2]
# break field = city[:i]
# break field = city[5] (10,2)
# break field = :field[:i] (:x, :y)
```

Remember that variables used to identify the array element or the substring must be colon–expanded. The following are not legal break field designations:

```
# break field = city[i]
# break field = :field[i] (a, b)
```

header

The `header` clause specifies the processing to be performed and the area(s) to be printed at the beginning of a break group. Typical header processing includes re–initializing subtotals for the new break group and printing a header area with the current break field value, column titles, etc.

Each `header` clause must include at least one `area`, `report`, `jpl`, or `call` subclause.

`area` The `area` subclause specifies a report area to be output at the beginning of a break group. The `area` subclause is not required, as long as at least one `report`, `jpl`, or `call` subclause is present in the `header` clause; multiple `area` subclauses are permitted.

`report`
The `report` subclause specifies a subreport to be invoked at the start of each break group. This subclause is optional, as long as at least one `area` , `jpl`, or `call` subclause is present; multiple `report` subclauses are permitted. Refer to Section 6.2 for a complete discussion of subreports.

`jpl` The `jpl` subclause specifies a JPL procedure to be run at the start of each break group. This subclause is optional, as long as at least one

area , report, or call subclause is present; multiple jpl subclauses are permitted.

call The call subclause specifies a C (or other supported language) routine to be run at the start of each break group. This subclause is optional, as long as at least one area , report,or jpl subclause is present; multiple call subclauses are permitted.

area, report, jpl, and call subclauses are executed in the order they are encountered in the header clause.

The area subclause in the header clause can also contain the following keywords to control output of the area:

nodupl

The nodupl keyword causes header output to be suppressed if it immediately follows the header output of the next higher break level. All other subclauses associated with the header clause are executed, however.

If this keyword is present in the highest–level break header, the report compiler issues a warning and ignores the nodupl.

The nodupl keyword is typically used in the case where columns labelled by the lower–level header are a subset of those labelled by the higher–level header. In such a case, it is redundant to output the lower–level header if the same (and likely, more) column labels appear immediately above.

Figure 12 illustrates such an application. A small portion of the report script and the relevant report areas are shown.

shrink

The shrink keyword removes excess blank lines from the output area. Refer to Section 6.5.1 for information on sizing report areas dynamically.

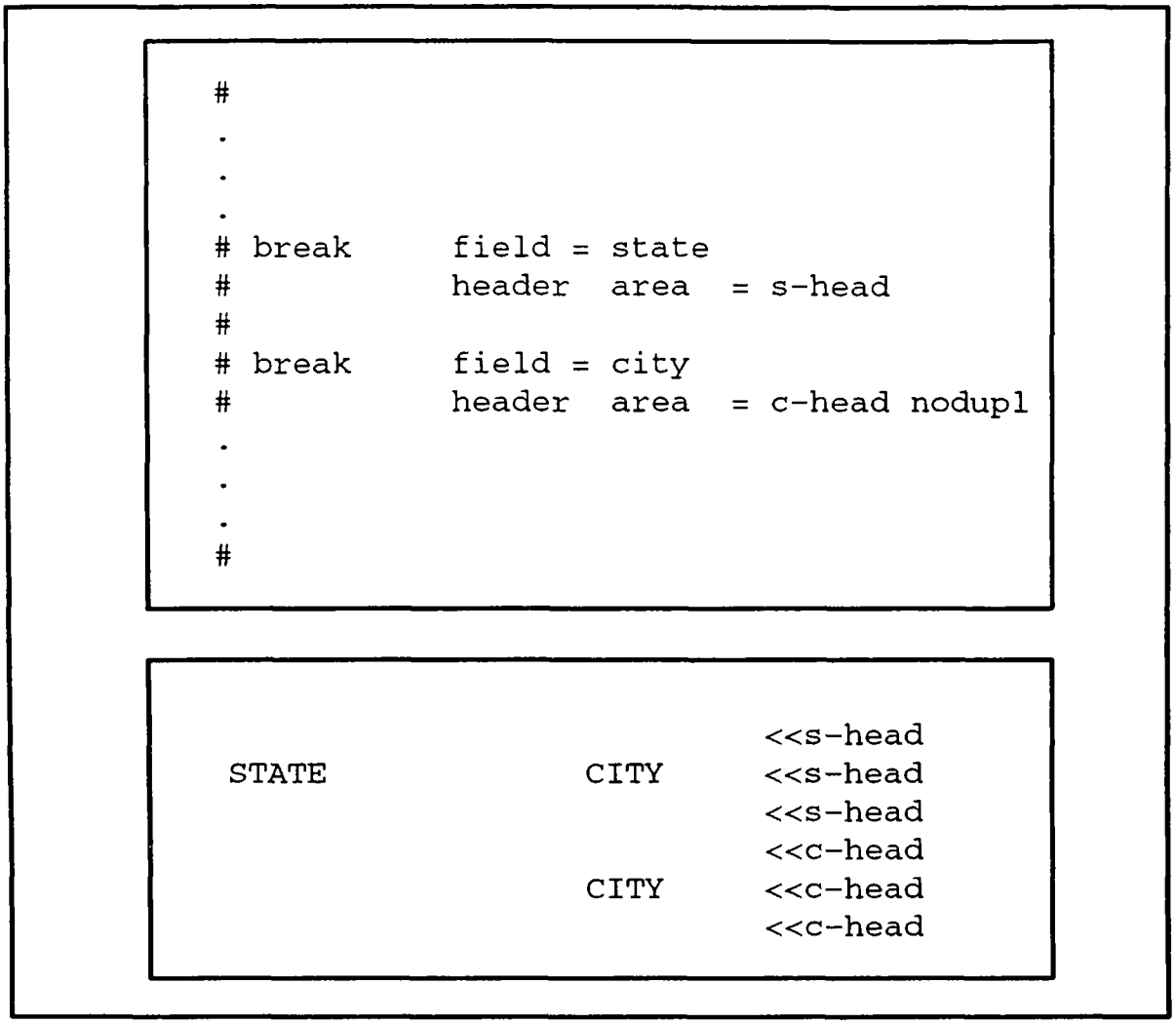
split

The split keyword allows the area to begin, end, or span a page. If this keyword is not present, ReportWriter will keep all headers together and unbroken.

showattop

If the showattop keyword is present, the area will appear at the top of each new page during detail processing (after the page header), whether or not a break has occurred at that point.

The nodupl and showattop keywords can be combined for a break header area. In the event of a conflict, nodupl supersedes showattop.That is, showattop directs



```
#
.
.
.
# break      field = state
#            header  area  = s-head
#
# break      field = city
#            header  area  = c-head nodupl
.
.
.
#
```

```
                                  <<s-head
STATE                 CITY        <<s-head
                                  <<s-head
                                  <<c-head
                      CITY        <<c-head
                                  <<c-head
```

Figure 12: A Typical Use of the `nodupl` Keyword

that the break header area be output at the top of each page; if however, the next higher-level header area is output at that point (whether because a break has occurred or because it, too, has the `showattop` keyword), `nodupl` suppresses the current break header output.

footer

The `footer` clause specifies the processing to be performed and the area(s) to be printed at the end of a break group. This is typically where subtotals for the group are printed, before the next group begins.

Each `footer` clause must include at least one `area`, `report`, `jpl`, or `call` subclause. In addition, the `noorphanbreak` keyword can be included as necessary to control formatting of the break footer.

`area` The `area` subclause specifies a report area to be output at the end of a break group. The `area` subclause is not required, as long as at least one `report`, `jpl`, or `call` subclause is present in the `footer` clause; multiple `area` subclauses are permitted.

`report`
The `report` subclause specifies a subreport to be invoked at the end of each break group. This subclause is optional, as long as at least one `area` , `jpl`, or `call` subclause is present; multiple `report` subclauses are permitted. Refer to Section 6.2 for a complete discussion of subreports.

`jpl` The `jpl` subclause specifies a JPL procedure to be run at the end of each break group. This subclause is optional, as long as at least one `area` , `report`, or `call` subclause is present; multiple `jpl` subclauses are permitted.

`call` The `call` subclause specifies a C (or other supported language) routine to be run at the end of each break group. This subclause is optional, as long as at least one `area` , `report`,or `jpl` subclause is present; multiple `call` subclauses are permitted.

`area`, `report`, `jpl`, and `call` subclauses are executed in the order they are encountered in the `footer` clause.

Suppose, for example, that part of the footer processing consists of clearing out the subtotals from the previous break group. The `area` subclause must appear before the `jpl` or `call` that invokes the procedure to zero out these variables. Otherwise, the footer area will display the re–initialized values rather than the correct totals for the group. In the following example, `footcalc.jpl` performs some calculations and additional processing to generate values for the footer area; `cleartot.jpl` zeros out the totals from the break group just completed.

```
# break     field   = custname
#           footer  jpl     = footcalc
#                   area    = custacct
#                   jpl     = cleartot
```

The `area` subclause in the `footer` clause can also contain the following keywords to control output of the area:

`nodupl`
If the `nodupl` keyword is present in an `area` subclause the `footer` clause and if the next higher break level occurs at the same time, this footer area is not output. All `report`, `jpl`, or `call` subclauses associated with the `footer` clause are executed, however.

If this keyword is present in the highest–level break footer, the report compiler issues a warning and ignores the `nodupl`.

`shrink`

The `shrink` keyword removes excess blank lines from the output area. Refer to Section 6.5.1 for information on sizing report areas dynamically.

`split`

The `split` keyword allows a footer area to begin, end, or span a page. If this keyword is omitted and if there is not sufficient room on the current page to fit the entire area, it begins on a new page.

The `footer` clause can also contain the following keyword to control output of all specified areas:

`noorphanbreak`

If this keyword is present and if the group just printed had only one member, the footer areas are not printed and subreports are not invoked. All `jpl` and `call` subclauses associated with the `footer` clause are executed, however.

The `noorphanbreak` keyword can be qualified by the `lines` subclause. `lines` specifies a number of blank lines to be output in lieu of the footer areas.

For example, to omit the footer area when a break group consists of just a single row and to insert three blank lines in place of that area, include the following in the `footer` clause:

```
# noorphanbreak    lines = 3
```

newpage

Specify `newpage` if you want each break group at this level to begin on a new page. A new page is begun after processing of break footer output for all break levels.

The presence of the `newpage` keyword at a given level implies the presence of `newpage` at all higher levels. If `newpage` is specified at multiple levels in the break hierarchy, only one page break occurs, no matter how many levels have broken.

norepeat

The `norepeat` keyword suppresses output of the break field in the detail area except for the first fetched row in each break group. The field is also output, however, when the row fetched is the first one on a page.

Consider, for example, the following report:

Vendor	Category	Part Number	Units on Hand
ABC Mfg.	bolt	12-421	179
ABC Mfg.	bolt	12-422	220
ABC Mfg.	bolt	12-583	112
ABC Mfg.	bolt	12-593	381
ABC Mfg.	nail	19-635	10000
ABC Mfg.	nail	19-640	5890
ABC Mfg.	nail	19-735	8000
ABC Mfg.	nut	22-421	203
ABC Mfg.	nut	22-422	190
ABC Mfg.	nut	22-593	380
ABC Mfg.	nut	22-601	512

ABC Mfg. Total Units on Hand: 26067

To make this report less redundant and more readable, include the `norepeat` keyword in the vendor field `break` statement. The field that appears in the category column can also be defined as a break field so that its contents will be printed only when the value changes. The `break` statements for this report might look like:

```
# break     field  = vendor
#           header jpl     = cleartot
#                  area    = vendhead
#           footer jpl     = vendcalc
#                  area    = vendfoot
#           norepeat
#           newpage
#
# break     field  = category
#           norepeat
```

The `newpage` keyword in the first `break` statement ensures that each vendor's listing will begin on a separate page.

These break statements yield a report that looks like:

Vendor	Category	Part Number	Units on Hand
ABC Mfg.	bolt	12-421	179
		12-422	220
		12-583	112
		12-593	381
	nail	19-635	10000
		19-640	5890
		19-735	8000
	nut	22-421	203
		22-422	190
		22-593	380
		22-601	512
ABC Mfg.	Total Units on Hand:		26067

norepeatattop

The norepeatattop keyword suppresses output of the break field in the same manner as norepeat, with the following difference: if the listing for the break group spans multiple pages and if norepeatattop is specified, the break field is not repeated in the first row on each new page, as it would be if norepeat had been specified instead.

If the norepeatattop keyword is specified, the break field is output in the detail area only when its value changes.

5.3
OUTPUTTING A SINGLE AREA OR INVOKING A PROCEDURE: insert

Use the insert statement to output a report area, invoke a subreport, and/or to specify procedure(s) to be executed. Any area output or procedure invoked with this statement is performed once only. (Contrast this with the detail statement, which is driven by a database query and performs area output and associated processing for each fetch.)

Typical uses for the insert statement include

- producing a title page for the report,
- generating a trailer page showing grand totals accumulated during report generation,

- forcing a page break so that the next report area will begin on a new page, or
- invoking a JPL procedure that performs any necessary cleanup at the end of a report, including closing connections to the database.

Each `insert` statement must include at least one `area`, `report`, `jpl`, or `call` clause or the keyword `newpage`. In addition, the keywords `split` and `shrink` can be included as necessary to control formatting of this report area.

A typical `insert` statement might look like:

```
# insert   area    =   title2
#          jpl     =   pagenum
#          jpl     =   "init"
#          newpage
```

5.3.1

Insert Processing

When an `insert` statement is encountered in the report script, ReportWriter executes its `area`, `report`, `jpl`, and `call` clauses in order of appearance. Thus, any procedures that compute values required for the output area must be invoked with `jpl` or `call` clauses placed before the `area` clause. Similarly, if any procedures modify data in ways that you do not want reflected in the output area, place the corresponding `jpl` or `call` clauses after the `area` clause.

5.3.2

`insert` Clauses and Keywords

`area`

The `area` clause specifies a report area to be output. This clause is optional; multiple `area` clauses are permitted in an `insert` statement.

The `area` clause can be modified by the `shrink` keyword. If the specified area contains any arrays that are not fully populated, `shrink` removes unused elements at the end of the array and shrinks the output accordingly. The `shrink` keyword should be placed after the area name:

```
# area =   summary shrink
```

Refer to Section 6.5.1 for more information on sizing areas dynamically.

report

The `report` clause specifies a subreport to be invoked. This clause is optional; multiple `report` clauses are permitted.

Refer to Section 6.2 for a complete discussion of subreports.

jpl

The `jpl` clause specifies a JPL procedure to be executed. This clause is optional, and multiple `jpl` clauses are permitted.

call

The `call` clause specifies a C (or other supported language) routine to be executed. This clause is optional, and multiple `call` clauses are permitted.

newpage

The `newpage` keyword forces a page break.

If the `insert` statement contains any `area` or `report` clauses, the page break occurs after these area are output.

If no `area` or `report` clause is present, `newpage` simply forces a page break. This is useful for ensuring that the next area output begins on a new page. For example, suppose a detail section of the report has just been completed and the next area to be output is a title page for a new report section. The following script fragment shows how to make sure that the title occupies a page of its own:

```
# /* The following statement forces a page break,
# ensuring that the next area output will begin on
# a new page: */
#
# insert   newpage
#
# /* When the title page for section 2 is output, use
# the newpage keyword to prevent the printing of any
# additional area on the same page. */
#
# insert   jpl     =   sec2init.jpl
#          area    =   sec2titl.jam
#          newpage
```

`newpage` does not force blank pages. If a page is currently opened, it is closed so that the next output will be on a new page. If there is no open page, `newpage` has no effect; that is, it will not cause a blank page to be output.

If you need to force a blank page, create a report area consisting of a blank line and use `newpage` to ensure that it occupies a page of its own, as in the following:

```
#
# insert newpage /* next area output will be on a
#                   new page */
#
# insert area  = empty
#                newpage
#
```

`split`

By default, ReportWriter will not unnecessarily split a multi–line area between pages. If there is not sufficient room on the current page to accommodate the entire area, ReportWriter will force a page break and begin the area on a new page.

The `split` keyword overrides this pagination rule, permitting the area to begin part way down on a page, even if that means that the area will be split across pages.

The `split` keyword must be placed with the `area` clause to which it applies.

5.4
INITIALIZING THE REPORT: `init`

The `init` statement allows you to initialize the page size and left margin parameters for the report as well as to specify the arguments accepted by the report on invocation.

In addition, you can use the `init` statement to output report areas and/or to specify procedure(s) to be executed as part of report initialization. This is usually the appropriate place to invoke procedures that initialize report variables and, if you are running ReportWriter stand–alone, open a connection to the database.

The `init` statement can include any of the following clauses and keywords: `lines`, `columns`, `leftmargin`, `feedlines`, `area jpl`, `call`, `fixedlength`, `varlength`, `parameter`, `newpage`, `shrink` and `split`.

A typical `init` statement might look like:

```
# init      jpl     = init_rpt_var
#           jpl     = open_db_conn
#           area    = titlepg
#           newpage
#           lines       =   54
#           columns     =   72
#           leftmargin  =   8
```

NOTE: When ReportWriter is run from a JAM/DB*i* application, it is usually the application, not ReportWriter, that opens the connection to the database. When ReportWriter is run from the stand-alone utility `rwrun`, however, the report, itself, must be responsible for connecting to the database.

5.4.1
Initialization Processing

The `init` statement is normally the first executable statement in the report script. The `area`, `jpl`, and `call` clauses are executed in order of appearance (as in any other statement where these clauses can be used). The `lines`, `columns`, `feedlines`, and `leftmargin` clauses and the `fixedlength` and `varlength` keywords govern page size and formatting until overridden by the presence of another `init` statement in the script.

Values for `lines`, `columns`, `feedlines`, and `leftmargin` can also be specified in the device file, as can the `fixedlength` or `varlength` keyword. If specified in both the device file and in an `init` statement, the value in the `init` statement takes precedence. If not specified in either place, ReportWriter default values are used. Refer to Section 5.4.4 for a discussion of the ReportWriter defaults.

Output parameters specified in the `init` statement take effect when the statement is encountered in the script. Therefore, if one or more `insert` or `detail` statements precede the first `init` statement in the script, any areas output as a result of those statements will be on pages governed by parameters in the device file, if present, or by ReportWriter defaults.

Since output parameters in the `init` statement take precedence over the corresponding entries in a device file, it is recommended that you use the `init` statement only for output parameters that should never change, no matter where the report output is directed. Specify all other parameters in device files, each of which can be tailored to a particular output device or file. Refer to Section 8.1 for further information on device configuration files.

Arguments passed to the report on invocation must be defined in `parameter` clauses in the `init` statement

5.4.2
`init` Clauses and Keywords

The `area`, `jpl`, and `call` clauses and the `split`, `shrink`, and `newpage` keywords function exactly as they do in the `insert` statement. Refer to the `insert` statement description in Section 5.3.1 for an explanation of these clauses and keywords.

lines

Use the `lines` clause to specify the report page length.

The value specified in this parameter refers to the space where you want printing to occur on the page. It includes the areas for the page header and footer as well as the space available for the report body.

Suppose you are designing a report to be printed on standard 8.5 by 11 inch paper. At 6 lines per inch, the paper can physically accommodate 66 lines. Suppose, too, that you want to leave top and bottom margins of one inch each, allowing 54 lines per page for the body of the report. The page header and footer you have created will each require two lines within the margin space (perhaps one printing line and one blank line each to separate it from the report content), leaving top and bottom unprinted areas of four lines each[2]. To obtain this page length, specify

```
# init lines   = 58
```

To produce a report without page breaks, set the value of `lines` to zero:

```
# init lines   = 0
```

The default page length is 60.

columns

Use the `columns` clause to specify the width of the report page.

The value specified in this parameter refers to the area of the page from the leftmost printable position to the rightmost column in which you want printing to occur. It includes the left margin plus the area in which the report will be printed but does not include the right margin. The example given in the description of the `leftmargin` clause, below, shows how these two parameters, together, position the report horizontally on the page.

The default value for `columns` is 132.

leftmargin

Use `leftmargin` to specify a number of spaces to be prepended to each non–blank line. The spaces used to create the left margin must be included in the page width (`columns` clause).

Suppose, for example, you are defining a report that will print on an 80–column printer. You want margins of 8 characters on either side, for a print area 64 columns wide. To achieve this effect, specify

2. ReportWriter does not have an explicit way to control page top and bottom margins. The physical page size in lines minus the lines parameter determines how many blank lines constitute the top and bottom margins, combined. Alignment of the page in the printer determines how these blank lines are apportioned.

```
# init  columns = 72    /* allows up to 72 characters
#                          from the leftmost printable
#                          position */
#
#      leftmargin = 8 /* forces a left margin of 8 char.;
#                        report printing begins at
#                        position 9, leaving 64
#                        characters per line for printing
#                        of the report text */
```

If no `leftmargin` value is specified, it is assumed to be zero.

feedlines

The `feedlines` parameter specifies the number of line feed characters that should be used to separate pages. If this parameter is omitted or if the value specified is 0, ReportWriter uses a form feed to begin the next page. If this clause is used, the value of `feedlines` plus `lines` must equal the physical length of the page.

To continue the example provided above in the discussion of `lines`, if the printer requires an explicit number of line feed characters rather than a form feed to position printing on a new page, include a `feedlines` clause in the `init` statement:

```
# init  lines      =   58
#       feedlines  =   8
```

fixedlength

If the `fixedlength` keyword is specified, all report lines are padded with spaces to equal the number of columns specified. If `fixedlength` is not specified, ReportWriter outputs variable length lines.

varlength

If the `fixedlength` keyword appears in the device file, you can override it by including `varlength` in the `init` statement. Since variable length output lines is the default, this keyword is not normally required, except to ensure that line output is variable length, no matter what the device file in use specifies.

The keywords `fixedlength` and `varlength` are mutually exclusive.

parameter

Use a parameter clause to specify each argument to be accepted by the report when it is invoked. Refer to Sections 5.4.3 and 6.3 for information on passing arguments to reports and subreports.

5.4.3

Accepting and Processing Arguments

All arguments to be accepted by a report must be declared in parameter clauses in the report's init statement. The syntax of the parameter clause is

parameter = ***name***

where:

name is the JAM variable to receive the value of the next unprocessed argument in the invocation string or in RWOPTIONS. If all arguments have been exhausted, the value of ***name*** remains unchanged.

Use a separate parameter clause for each argument accepted by this report. The order of the parameter clauses determines the order in which arguments must be passed to the subreport.

Each parameter must exist as a field on the report format screen or in the LDB. Any number of parameter clauses can appear in an init statement.

The report in the following example accepts two arguments. The first is used in the WHERE clause of the detail query. The other specifies the report area to use in the page header.

```
# << begin report >>
#
# init jpl =   startup
#      parameter  =   parm1
#      parameter  =   parm2
#
# page header area    =   :parm2
#      footer jpl     =   pnum
#             area    =   pfoot
#
# break field  =   custno
.
.
.
# detail   query   "SELECT * FROM orders \
#                   WHERE sales_id = ':parm1'\
#                   ORDER BY cust_no"
.
.
.
# << end report >>
```

If arguments to a primary report or subreport appear both in `RWOPTIONS` and in the invocation string, those in `RWOPTIONS` are passed first.

5.4.4

Output Parameter Defaults

As noted above, ReportWriter defaults are used for any page specification parameters not supplied in either the `init` statement or in the device file. These defaults are:

`lines` and `columns`
: The default page size is 60 lines by 132 columns.

`leftmargin`
: If no value is specified for the left margin offset, it is assumed to be 0.

`feedlines`
: If the `feedlines` parameter is not specified, ReportWriter issues a form feed to begin each new page.

`fixedlength` and `varlength`
: In the absence of either of these keywords, ReportWriter defaults to variable–length output lines.

5.5 SPECIFYING PAGE HEADERS AND FOOTERS: page

The page statement allows you to specify headers and footers to appear on each page of the report.

The header and footer specified in a page statement are distinct from those specified in a break statement. Break headers and footers are displayed when a break group begins or ends, which may not necessarily correspond to the start or end of a page. Page headers and footers, on the other hand, are related strictly to the beginning or end of a page, and, if specified, appear on every page, regardless of the report content.

Each page statement can specify a page header, a page footer, or both.

A sample page statement might look like:

```
# page  header  jpl     =   pagenum
#               area    =   pghead
#       footer  area    =   "footer"    float
```

5.5.1 Page Break Processing

Page specifications take effect at the point where the page statement is encountered in the report script and remain in effect until a clear, clear pagespecs, or another page statement is encountered.

The order of header and footer clauses within the page statement is not significant. The order of area, jpl, and call subclauses within the header and footer clauses, however, is significant and determines the order in which processing and area output occur.

5.5.2 Changing Page Specifications

ReportWriter provides several ways to change page header and footer information.

The most straightforward method is to use clear or clear pagespecs to cancel the current page specifications. If a new header and/or footer is required, place a new page

statement after the `clear`. This method completely cancels out all previous page header and footer specifications and ensures that there are no unintended interactions between the `page` statements.

Page specifications can also be changed by the presence of a subsequent `page` statement. In the absence of any `clear` statements, however, this method can result in interactions between the `page` statements. To avoid unwanted effects from these interactions, keep in mind the way `page` statements supersede all or part of the current specification:

- Any header or footer in the new `page` statement supersedes the corresponding (header or footer) specification currently in effect from a prior `page` statement.
- If a header or footer clause is omitted in the new `page` statement, the corresponding component (if any) currently in effect remains in effect.

Consider the following examples:

1. The page specification currently in effect includes both a header and a footer. At this point in the report, it is necessary to change just the page header; the footer should remain as currently specified.

   In this situation, the interaction between `page` statements can work to the developer's advantage. At the point in the script where you want the page specifications to change, enter a `page` statement containing only a `header` clause. From this point on, the new header will appear at the top of each report page; the footer previously in effect will continue to appear at the bottom of each page.

2. The page specification currently in effect contains only a footer. The next section of the report should be displayed on pages with a header only and no footer.

   In this situation, you must use the `clear` or `clear pagespecs` statement to cancel out the previous page specification before entering the new `page` statement with the desired header specification. Otherwise, the previous footer area and processing will remain in effect.

3. The page specification currently in effect contains a header with both a JPL procedure and an area specified. The next section of the report will also require processing at the beginning of each page but should not have an area output.

   In this case, the `page` statement alone can override the previous page specification. When the second `page` statement is encountered with its `header` clause, the entire previous header specification is superseded. Simply the presence of a `header` clause serves to override any previous header specification; the subclauses do not have to cancel each other out individually.

If a page has been started but not closed when the page specifications change (either as a result of a new `page` statement or a `clear` or `clear pagespecs` statement), that page remains governed by the prior page specifications. When filled, it will be closed with the previous page footer, since it was opened with the previous header. The new page specifications will take effect when a new page is started.

A `page` statement does not imply `newpage`. To force the new page specifications to take effect immediately, issue an `insert newpage` just before or after the `page` statement. That will force the currently open page to be closed (with the old page footer) and any subsequent output to appear on a new page. Section 5.6.1 contains an example showing a typical scenario in which the detail area of the report, with page headers and footers, is to be followed by a separate trailer page with no headers or footers.

5.5.3

`page` Clauses and Keywords

`header`

The `header` clause specifies the processing to be performed and the area to be printed at the beginning of each page. Typical header processing might include updating the page number or capturing the current value of a break field for inclusion in the page header.

Each `header` clause must include at least one `area`, `report`, `jpl`, or `call` subclause.

`area` The `area` subclause specifies the report area to be output at the beginning of each page. The `area` subclause is not required, as long as at least one `report`, `jpl`, or `call` subclause is present in the `header` clause; multiple `area` subclauses are permitted.

The `area` subclause can be modified by the `shrink` keyword, if required. Refer to Section 6.5.1 for information on sizing report areas dynamically.

`report`
The `report` subclause specifies a subreport to be invoked at the beginning of each page. This subclause is optional, as long as at least one `area` , `jpl`, or `call` subclause is present; multiple `report` subclauses are permitted. Refer to Section 6.2 for a complete discussion of subreports.

`jpl` The `jpl` subclause specifies a JPL procedure to be run at the beginning of each page. This subclause is optional, as long as at least one `area` ,

report, or call subclause is present; multiple jpl subclauses are permitted.

call The call subclause specifies a C (or other supported language) routine to be run at the beginning of each page. This subclause is optional, as long as at least one area , report,or jpl subclause is present; multiple call subclauses are permitted.

area, report, jpl, and call subclauses are executed in the order they are encountered in the header clause.

footer

The footer clause specifies the processing to be performed and the area to be printed at the end of each page. Processing associated with the footer might include calculating the page number or computing or capturing any other data you want to appear at the bottom of the page.

Each footer clause must include at least one area, report, jpl, or call subclause.

area The area subclause specifies the report area to be output at the end of each page. The area subclause is not required, as long as at least one report, jpl, or call subclause is present in the footer clause; multiple area subclauses are permitted.

The area subclause can be modified by the shrink keyword, if required. Refer to Section 6.5.1 for information on sizing report areas dynamically.

The area subclause can also be modified by the float keyword, described below.

report

The report subclause specifies a subreport to be invoked at the end of each page. This subclause is optional, as long as at least one area, jpl, or call subclause is present; multiple report subclauses are permitted. Whenever a report is invoked from a page footer, the reservelines subclause must be included to specify the number of lines the subreport will occupy. Refer to Section 6.2 for a complete discussion of subreports.

jpl The jpl subclause specifies a JPL procedure to be run at the end of each page. This subclause is optional as long as at least one area, report, or call subclause is present; multiple jpl subclauses are permitted.

`call` The `call` subclause specifies a C (or other supported language) routine to be run at the end of each page. This subclause is optional as long as at least one `area`, `report`, or `jpl` subclause is present; multiple `call` subclauses are permitted.

`area`, `jpl`, and `call` subclauses are executed in the order they are encountered in the `footer` clause.

`area` subclauses in the `footer` clause can also contain the following keyword to control placement of that area:

`float` If this keyword is present, the area immediately follows the last printed line on the page. Otherwise, it appears at the bottom of the page. All areas with the `float` designation must be output before any non-floating areas.

The following example shows how both floating and non–floating areas can be combined in a page footer.

```
# page  footer  area = pf1 float
#               jpl  = j3
#               area = pf2 float
#               jpl  = j4
#               area = pf3
```

5.6
CANCELLING PAGE AND BREAK SPECIFICATIONS: `clear`

The `clear` statement cancels all previous page and/or break specifications. This statement is processed as it is encountered in the report script, cancelling specifications before it in the script, but not affecting those placed after it.

The keywords `breakspecs` and `pagespecs` can, optionally, be included to indicate that only the break or page specifications are to be cancelled.

If neither keyword is present, `clear` cancels all previous `page` and `break` statements. To cancel both page and break specifications at once, enter the following statement into the report script:

```
# clear
```

5.6.1

clear Keywords

breakspecs

If the breakspecs keyword is present, the clear statement cancels all break processing enabled by previous break statements.

To clear the break hierarchy without affecting the current page specifications, enter the following into the report script:

```
# clear breakspecs
```

Break specifications are cancelled only by the clear or clear breakspecs statement. Break specifications must be cleared if you are defining a new break hierarchy (for example, if rows are to be fetched from a second table using a new detail statement); otherwise, any new break statement would define a break subordinate to those already in effect.

pagespecs

If the pagespecs keyword is present, the clear statement cancels page header and footer specifications currently in effect. Further pages of the report will show no page headers or footers unless another page statement follows.

To clear the current page specifications without affecting the current break hierarchy, enter the following into the report script:

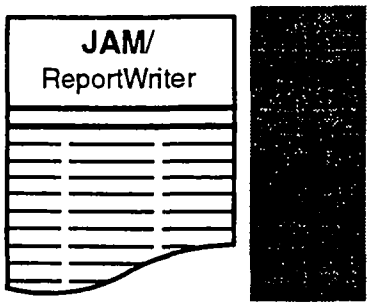
```
# clear pagespecs
```

Page specifications are cancelled by the clear or clear pagespecs statement. (In some circumstances, it is desirable to use a page statement not preceded by a clear to override current page specifications. Refer to Section 5.5.2 for more information on changing page specifications.)

The following example shows a typical context in which the clear statement is used to cancel previous page headers and footers:

```
# /* The detail area of this report is printed on pages
# with headers and footers specified in a "page"
# statement. */
#
# page          header area = pghead
#               footer area = pgfoot
# break         footer jpl = breakcalc
#                      area = breakarea
#
```

```
# insert   newpage /* forces detail area output to
#                     begin on a new page with
#                     the header and footer specified
#                     in the page statement above */
#
# detail       query   = "SELECT ..."
#              jpl     = dtljpl
#              area    = dtlarea
#
# /* End the report with a trailer page (area name =
# trailer) that has no headers or footers: */
#
# clear    pagespecs
#
# insert   newpage /* forces next output to begin
#                     on a new page */
#
# insert   jpl     = grandtot
#          area    = trailer
```



Chapter 6
Report Components

6.1
REPORTWRITER VARIABLES

6.1.1
Colon Expansion

ReportWriter allows the use of JAM colon–expanded variables in the report script. This permits you to design reports that are responsive to data that may not be known until runtime or that may vary during the course of generating the report.

NOTE: While colon–expanded variables are permitted in report script statements, they cannot be used in the `include screen` compiler directive.

Colon Substitution in Detail Queries

JAM variables can appear in the `query` clause of the `detail` report script statement. A colon must immediately precede the variable name, and the current value of that variable is inserted when the `query` is executed. In the following example, the value to match in the `name` column is not known until runtime.

```
# detail   query   = "select * from t\
#                     where name =':x'"
```

This feature is particularly useful when ReportWriter is invoked from within a JAM application. In the above example, x might be a JAM screen field whose value is entered by the end user. Alternatively, the values of the colon–expanded variables in the `query` clauses of a report script might be determined by some other processing performed by the application or by data previously fetched from the database.

Dynamic Reports

JAM variables can also be used to specify report areas, subreports, programs, or break fields in the report script. Any of the following report script clauses can accept a colon–expanded variable in place of a specific area, routine, or field name:

```
field
area
report
jpl
call
```

Substitution is fully dynamic: it is performed whenever the referenced object is needed. For example, suppose a report script contains the statement

```
# page header area    = :pghead
```

ReportWriter examines the current value of the variable `pghead` each time a page header is to be output. The value of this variable specifies the area on the report format screen to use as the page header. If the value of `pghead` is null when a page header is to be output, the header area is omitted for that page. If the value of `pghead` is not null but does not reference an area on the report format screen, a runtime error occurs.

Similar rules govern the use of colon–expanded variables in `field`, `jpl`, and `call` clauses. When the field, JPL procedure, or C routine name is needed at runtime, it is determined from the current value of the applicable variable.

If the resulting string is null in the case of a `jpl` or `call` clause, no program is called. If a `field` specification is null, its entire `break` statement is ignored. A runtime error occurs if variable substitution yields a non–null object that cannot be found.

Quotation Marks around Colon–Expanded Variables

A colon–expanded variable must be surrounded by quotation marks if the resulting string would have to be enclosed in quotes. In the following `query` statement,

```
# detail    query    = "select * from table1\
#                       where name = ':who'"
```

suppose that the variable who contains the text string Shakespeare, the name you want to search for in the database. ReportWriter will pass to JAM/DB*i* the string

```
select * from table1 where name = 'Shakespeare'
```

If the quotation marks around :who were omitted in the query, ReportWriter would have passed to JAM/DB*i*

```
select * from table1 where name = Shakespeare
```

In this case, Shakespeare would be treated as a column name, rather than a character string.

Users of JAM/DB*i* release 5 may use the colon–plus (:+) form of colon expansion in preference to enclosing string–valued variables in quotes. Refer to the JAM/DB*i* documentation for information colon–plus expansion.

Quotation marks are also required when the resulting string contains (or might contain) spaces or special characters. Suppose the argument to a jpl clause is a colon–expanded variable that is likely to contain both the name of a procedure to invoke and one or more arguments to be passed to the procedure. The variable to be expanded would have to be enclosed in quotes since there is the possibility that the resulting string would need quotes.

```
detail  jpl = ":proc_and_args"
```

When in doubt as to whether or not the expanded variable will result in a string that must be quoted, use the quotation marks. Refer to Section 4.2.2 for more information on when quotation marks are required in the report script.

Variable Substitution For Numeric Values

Colon–expanded variables can also be used in the report script in place of numbers. For example:

```
# init      lines   = 3
# init      lines   = :x
```

All variables substituted for numeric values must be colon–expanded. The following are not legal ReportWriter statements:

```
# init      lines   = 3x
# init      lines   = x
# init      lines   = ":x[2] (1,3)"
```

Colon–expanded variables cannot be used in device configuration files.

6.1.2
Scope of Variables

Any variable that appears in the report script, whether as a colon–expanded variable or as the break field, must be either a field on the report format screen or an LDB variable.

Following good **JAM** and **JAM/DB***i* programming practice, use the LDB judiciously. Where possible, use screen variables, instead.

Fields on the report format screen do not have to appear in output areas. Any field required for data passing only, rather than for report output, should be placed on a screen line with no area name tag.

Any field on the report format screen (including non–output fields) can be used as a variable in the report script or JPL. Refer to Section 4.1.4 for a further description of non–output fields.

6.2
SUBREPORTS

JAM/ReportWriter release 5.1 supports subreports that can be invoked directly from any other report.

In general, any report can be used as either a primary report or a subreport. There is no inherent distinction, since the same report format screen requirements and script capabilities apply, whichever way the report is used. A report is a *primary report* if it is invoked from the command line or from a JPL or C routine. A report is a *subreport* if it is invoked from another report.

ReportWriter provides you with considerable flexibility in developing both primary reports and subreports. A single report, for instance, can be designed to be invoked in either context, or you may choose to tailor it for one particular use. The comprehensive subreport example in Appendix D shows how a single report might be used as either a primary report or a subreport.

In addition, you can decide where you want to define your subreports. If the report will also be used as a primary report or may be invoked as a subreport from several different reports, you will want to create a separate report format screen for it. On the other hand, if it will only be invoked from one particular report, you might choose to define it as part of the invoking report's format screen.

The following sections describe how to define and invoke subreports. Two subreport examples are provided in Appendix D.

6.2.1

Prerequisites

The subreport capability described here applies only when ReportWriter is linked with

- JAM release 5.03a or higher and
- JAM/DB*i* release 5.

If you are using an earlier version of JAM or JAM/DB*i*, refer to Section 10.1 of this manual for an alternate means of producing subreports.

6.2.2

Defining the Subreport

As noted above, the subreport can be defined in a separate report format screen, can reside in the report format screen for the primary report, or can reside in a report format screen with other subreports.

If multiple reports are defined in the same report format screen, each must be surrounded by the `<<begin report>>` and `<<end report>>` compiler directives to indicate the beginning and end of each script. The syntax of the `<<begin report>>` compiler directive is:

```
<< begin report [= name] >>
```

where:

name	is the name by which the subreport is invoked; it is required for any report that is called as a subreport

The names of all reports defined and/or included in a report format screen must be unique in the first 21 characters.

If the first report defined in a report format screen does not have the ***name*** argument, it is known by the name of the screen and can be invoked only as a primary report. If you want to invoke it as a subreport, you must specify the ***name*** argument in the `<<begin report>>` compiler directive. This argument is optional only for the first report script associated with a screen; it is required for all subsequent scripts. Only the first script in a report format screen can be invoked as a primary report.

Each <<begin report>> must be paired with an <<end report>> compiler directive following the corresponding script. Nesting of report scripts is not permitted.

If you have subreports defined in report format screens external to the parent report, be sure to import these screens with the <<include screen>> compiler directive. When a report format screen is included in a compiled report, any report defined in the included screen can be invoked as a subreport. (Refer to Section 4.3.2 for more information on the <<include screen>> compiler directive.)

6.2.3

Invoking the Subreport

Use the report clause to invoke a subreport. This clause/subclause can appear as a clause in either of the following statements:

```
detail
insert
```

or as a subclause in the following clauses:

```
break header
break footer
page header
page footer
```

The syntax for the report clause/subclause is

```
report   =   invocation_string
             [preserve]
             [preserve breakspecs]
             [preserve initspecs]
             [preserve pagespecs]
             [reservelines = number]
```

where:

invocation_string

consists of the name of the invoked subreport followed, optionally, by arguments passed to the report. The arguments are processed by the parameter clauses in the subreport's init statement. If arguments are passed to the subreport, the entire invocation string must be enclosed in quotation marks; this is analogous to the way

JPL and C routines are invoked in the script, as described in Sections 4.2.2 and 6.4.1.

number this argument to the `reservelines` subclause indicates the maximum number of lines the subreport will occupy; `reservelines` is required for subreports invoked from page footers; it is optional elsewhere. This subclause is particularly useful for printing on forms with a fixed–length space for the subreport data.

If the `reservelines` subclause is present, ReportWriter assumes that the subreport will occupy the stated number of lines; page breaks are computed accordingly. ReportWriter does not attempt to merge leading and trailing blank lines of the subreport with those of the areas that precede and follow it. If the `reservelines` clause is omitted, ReportWriter attempts to consolidate blank lines with those of the adjacent areas (as described in Section 6.5.2).

If the `reservelines` subclause is present, the number of lines specified is always output exactly. If the subreport generates fewer lines than specified, ReportWriter pads the output with blank lines to achieve the required size. If the subreport generates more than the specified number of lines, the output is truncated, and an appropriate warning message is generated.

If the `reservelines` subclause is not given for a subreport invoked in a break header, the header may end the page as a "widow."

The `preserve` keywords cause the relevant specifications currently in effect for the "parent" report to take precedence over those in the subreport script:

`preserve breakspecs`
indicates that break specifications from the parent report remain in effect for the subreport; any breaks specified in the subreport are added to the existing hierarchy. By default, if the `preserve breakspecs` keyword is not present, the subreport begins with no break specifications in effect; any `break` statements in the subreport begin a new break hierarchy.

When invoked with `preserve breakspecs` in effect, the subreport also inherits the parent report's current break level to prevent unwanted duplication of break headers in the subreport. Refer to Section 6.2.4, "Preserving the Parent Report's Break Context," for more information on this topic.

`preserve initspecs`
tells ReportWriter to ignore the contents of the subreport's `init` statements, except for the `parameter` clauses; `init` specifications from the parent report remain in effect throughout the subreport. If this keyword is not present, the default is to begin with the parent report's initialization but to observe any `init` statements in the subreport.

Regardless of whether or not this keyword is present, `area`, `jpl`, `call`, and `parameter` clauses from the parent report's `init` statement are not executed in the subreport.

`parameter` clauses in the subreport's `init` statement are never overridden by the `preserve initspecs` keyword. They determine the arguments accepted by the subreport, whether or not this keyword is present.

`preserve pagespecs`
tells ReportWriter to ignore the contents of the subreport's `page` statements; page specifications from the parent report remain in effect. If this keyword is not present, the default is to begin with the parent report's page specifications but to observe any `page` statements in the subreport.

When a subreport is invoked in a page header or footer, the subreport does not inherit its parent's page specifications. The report compiler will ignore the `preserve pagespecs` keyword if it is applied to a subreport invoked from a page header or footer. Refer to Section 6.2.5 "Subreports Invoked from Page Headers and Footers," for more information on this topic.

Used without modification, the keyword `preserve` implies all three of the above variants.

`preserve` keywords used on subreport invocation also affect the corresponding `clear` statements in the subreport.

- If `preserve pagespecs` is in effect, `clear` or `clear pagespecs` does not reset the page specifications inherited from the parent.
- If `preserve breakspecs` is in effect, `clear` or `clear breakspecs` resets only that portion of the break hierarchy specified in the subreport. Breaks inherited from the parent report are retained.

6.2.4

Preserving the Parent Report's Break Context

Execution of a `detail` statement normally forces an initial break at all levels. If, however, a subreport inherits its parent's break specifications and if detail processing is already active in the parent, forcing initial breaks in the subreport would undesirably duplicate break output.

Therefore, if a subreport is invoked with the `preserve breakspecs` keyword in effect, the subreport inherits not only the parent's break hierarchy, but also the current break level. This ensures that when a `detail` statement is executed in the subreport, initial breaks are forced only for break levels subordinate to the parent report's current level and not for those at or above this level.

The type of statement from which the subreport (with `preserve breakspecs` in effect) is invoked determines which break levels are affected:

- A subreport invoked from the parent's `detail` statement never forces breaks within the parent's break structure.
- A subreport invoked from a break header or footer does not force a break at that or any higher level. It will force initial breaks at all lower levels.
- When invoked from an `insert` statement, the subreport inherits the break level current in the parent at the time the `insert` statement is processed. The `detail` statement in the subreport will force initial breaks at all lower levels. If no break level is active, then the `detail` statement will force breaks at all levels.
- When invoked from a `page` statement, the subreport inherits the break level current in the parent at the time the page header or footer is processed. The `detail` statement in the subreport will force initial breaks at all lower levels. If no break level is active, then the `detail` statement will force breaks at all levels.

If the subreport is not invoked with `preserve breakspecs` in effect, the parent's break context is ignored, and the subreport processes breaks according to the hierarchy in its own script.

6.2.5

Subreports Invoked from Page Headers and Footers

In most cases, a subreport begins by inheriting its parent's page header and footer. These page specifications may or may not be overridden by `page` statements in the subreport.

A subreport invoked from a page header or footer, however, does not inherit page specifications from the report that invoked it. If such a subreport were to inherit its parent's page specifications, it would attempt to invoke itself, resulting in an infinite loop.

The `preserve pagespecs` keyword is meaningless in the context of a subreport invoked from a page header or footer. If specified, it is ignored by the report compiler.

6.2.6

Storing Subreport Definitions in Separate Files

You may prefer to maintain both primary reports and subreports in a single screen to facilitate layout. On the other hand, modularity may be an important consideration for your application, and you may want to maintain your subreports in screens separate from the primary report. ReportWriter allows you to manage your report screens either way.

If you are maintaining all subreports with the primary report, you need only define them in the script as described above, in Section 6.2.2. If any of the subreports are maintained as separate JAM screens, you must use the `<<include screen>>` compiler directive to ensure that all such reports are included at compilation. (Refer to Section 4.3.2 for a description of the `<<include screen>>` compiler directive.)

6.2.7

Suppression of "No Rows Found" Warning Message

Subreports, by their very nature, may fail to fetch any rows. While this would be considered an error in a primary report and result in the "no rows found" warning message, it is not treated as an error in a subreport. Therefore, this warning message is never output for subreports.

6.2.8
Output Options in the Subreport: RWOPTIONS

If your subreport requires different output options from those set in the parent report, you should invoke a JPL or C routine that sets RWOPTIONS immediately before invoking the subreport. If the new value of RWOPTIONS differs from the old, the new value will govern the subreport's output. The change will not affect the parent report.

For example, suppose the primary report is invoked from a JPL routine, as in

```
cat RWOPTIONS "-o parentoutput"
rwrun mainrpt
```

The script and JPL code in mainrpt.jam might be something like

```
# << begin report >>
# insert area = hello
# detail   query   = ...
#          jpl     = setoptions
#          report  = "subreport :key"
# .
# .
# .
# << end report >>
#
# << begin report = subreport >>
# init     parameter = x
# detail   query   = ...
#          area    = a
#
# << end report >>

proc setoptions
cat RWOPTIONS "-a -o childoutput"
return 0
```

In the above example, all output from mainrpt, whether generated before or after the subreport is invoked, is sent to parentoutput. All output from subreport, when invoked as shown in the detail statement, is sent to childoutput.

6.3
REPORT ARGUMENTS

In ReportWriter release 5.1, both primary reports and subreports can accept arguments. They can be passed on invocation or through the LDB or environment variable RWOPTIONS.

NOTE: Command line invocation of a primary report does not access RWOPTIONS; report arguments must be included on the rwrun command line. For primary reports invoked within a JAM/ReportWriter application, arguments must be in RWOPTIONS. For subreport invocations, arguments can appear either in the invocation string or in RWOPTIONS.

6.3.1

Accepting and Processing Arguments

All arguments to be accepted by a report must be declared in parameter clauses in the report's init statement. The syntax of the parameter clause is

parameter = ***name***

where:

name is the JAM variable to receive the value of the next unprocessed argument in the invocation string or in RWOPTIONS. If all arguments have been exhausted, the value of ***name*** remains unchanged.

Use a separate parameter clause for each argument accepted by this report. The order of the parameter clauses determines the order in which arguments must be passed to the subreport.

Each parameter must exist as a field on the report format screen or in the LDB. Any number of parameter clauses can appear in an init statement.

The following report accepts two arguments. The first is used in the WHERE clause of the detail query. The other specifies a report area for the page header.

```
# << begin report >>
#
# init  jpl =   startup
#       parameter   =   parm1
#       parameter   =   parm2
#
# page  header  area    =   :parm2
#       footer  jpl     =   pnum
#               area    =   pfoot
#
# break field   =   custno
.
# detail    query   "SELECT * FROM orders \
#                    WHERE sales_id = ':parm1'\
#                    ORDER BY cust_no"
.
# << end report >>
```

If arguments to a primary report or subreport appear both in RWOPTIONS and in the invocation string, those in RWOPTIONS are passed first.

6.3.2

Passing Arguments to a Main Report

When a main report is invoked from the operating system command line with the rwrun command, report arguments are included in the invocation string. (RWOPTIONS is not accessed when a report is invoked from the command line.) The synytax for rwrun is

rwrun [-f|-a] [-o ***output***] [-d ***device***] ***report*** [***arg1 arg2 ...***]

The report arguments must appear in the order they are defined in the report's parameter clauses.

NOTE: If you want to preserve the command line syntax of ReportWriter release 5.0 (which considers extra tokens erroneous), edit the file rwopts.c according to the instructions in that file.

To invoke a primary report with arguments from within a JAM/ReportWriter application, place the arguments (in the required order) in the LDB or environment variable RWOPTIONS before invoking the report.

The following example shows how the report shown above might be invoked from the operating system command line:

```
rwrun myreport -o myoutput.txt fred region1
```

The report might be invoked from a JPL procedure in your JAM/ReportWriter application:

```
cat RWOPTIONS "fred region1"
rwrun myreport myoutput.txt
```

Note that in the above two examples, the argument values themselves, rather than variables, are passed to the invoked report. Alternatively, you might pass the arguments as colon–expanded variables:

```
cat arg1 "fred"
cat arg2 "region1"
cat RWOPTIONS ":arg1 :arg2"
rwrun myreport myoutput.txt
```

6.3.3

Passing Arguments to a Subreport

Pass arguments to a subreport either by including them in the invocation string of the `report` clause or by placing them in the LDB or environment variable `RWOPTIONS` before invoking the subreport.

Note that the `parameter` clauses in the subreport are always observed, even if the `preserve initspecs` keyword was specified when the subreport was invoked. (Parameter clauses from the parent report are not imported into the subreport.)

In the examples below, `arg1` and `arg2` are variables whose values are the arguments passed to the subreport.

The following example shows a subreport invocation with arguments.

```
# insert report = "myreport :arg1 :arg2"
```

In the next example, the arguments are placed in `RWOPTIONS` before the subreport is invoked.

```
# << begin report >>
.
.
.
# insert   jp1 =   setargs
#          report =   subr
# << end report >>
.
.
.
# <<begin report = subr>>
# init parameter   =   p1
#      parameter   =   p2
.
.
.
# << end report >>
proc setargs
cat RWOPTIONS ":arg1 :arg2"
return 0
```

In either case, the arguments must be listed in the order of the `parameter` clauses in the subreport.

6.4 FUNCTION CALLS

6.4.1 Passing Arguments

Most report script statements provide for the execution of JPL, C, or other supported language routines during report generation. JPL procedures are invoked with the `jpl` clause; C and other programming language routines, with the `call` clause.

To pass arguments to a procedure or routine, enclose the entire invocation string in quotation marks, as described in Section 4.2.2 of this manual. For example:

```
# jpl   = "condense_name namevar :last :first"
# call  = "inv_val :partno :list_price"
```

6.4.2 Using Return Codes

ReportWriter interprets return codes from procedures invoked with `jpl` or `call` clauses as follows:

`0` This is a normal return; it has no effect on report processing.

`-1` ReportWriter aborts with an error message.

any other negative value
: The report is closed prematurely at that point without producing an error message.

any positive value
: ReportWriter skips only the portion of the report currently being processed.

  When a positive value is returned from a `detail` statement, processing of the current fetch is curtailed, and the next fetch is begun.

  When a positive value is returned during break processing, the `query` is abandoned and the next report element is begun.

When a positive value is returned by a procedure in a `page`, `init`, or `insert` statement, all following area output and procedure execution for that statement is suppressed.

6.4.3

Calling C Routines

Any C routines required for processing your report are invoked through `call` clauses in the report script. These routines must be linked with the ReportWriter executables, `rwrun` and `jamrw`.

Follow the instructions in the JAM *Programmer's Guide* and in the provided files `funclist.c`, `rwmain.c`, and the makefile for installing user-developed functions. Refer to Section 4.5 for further information on this topic.

6.5

REPORT AREAS

6.5.1

Sizing Dynamically

In some cases, you cannot know in advance how much data will appear in a given report area. ReportWriter provides the keyword `shrink` to allow report designers to specify that a given report area should be condensed vertically if some of the allocated fields and trailing array elements are empty.

You might, for example, have an array that is not necessarily fully populated each time a fetch from the database is processed. Such would be the case if you are producing an employee report containing a salary and job description history for each employee. The number of lines required in the detail section will vary—from a line or two for relatively new employees, up to a dozen or more for those who have been with the company for many years. (This type of report is implemented as a sub-query. ReportWriter fetches the employees, and a procedure in the `detail` statement uses JAM/DB*i* to fetch the employee history. For more information on implementing sub-queries, refer to Section 10.1.)

Within a report area, you might also have certain fields and display text applicable to some, but not all, rows fetched from the database. Continuing with the personnel application example, a benefits report would require sections for health insurance and covered dependents, life insurance and beneficiaries, tuition assistance, and so forth. Since employees will not all be taking each of the available benefits, it would be nice to design the detail area so that, for each employee reported, the final report contains only the relevant fields.

Each of the above scenarios can be implemented through appropriate screen design combined with use of the `shrink` keyword in the ReportWriter script.

In the first example above, you might design the screen with several parallel arrays (date, salary, job description, etc.) to display the job history data. To provide sufficient space for reporting lengthy job histories without producing excess white space for those with short listings:

1. Make the parallel arrays large enough to accommodate the largest history you need to report. These must be onscreen arrays.
2. In the `area` clause of the `detail` statement, qualify the area name with the `shrink` keyword:

```
# detail     query   = "SELECT * FROM emp_table\
#                        ORDER BY emp_id, date"
#            jpl     = process_history
#            area    = emphist shrink
```

When the report is generated, each detail area will contain only as many history lines as needed for the employee reported.

The second example involves not only omitting unpopulated array occurrences, but also removing blank lines resulting from empty fields and unneeded display text. The screen design technique required here is to use data entry fields in lieu of display text to head up each section of benefits information. The JPL code executed before the area is output should populate these fields with the appropriate text if the associated benefit is to be reported and should leave the fields blank if the associated benefit is not taken. As long as no other field or display text is on the same line, use of the `shrink` keyword in the script will suppress printing of the line(s) if the associated benefit is not reported. (An example using this technique is shown in Section 10.1.)

Null fields on a **JAM** screen are not the same as blank fields. The `shrink` keyword removes only blank fields. Null fields are not removed unless the null character is a blank space.

The `shrink` keyword does not affect blank lines that contain neither fields nor display text.

6.5.2
Consolidation of Leading and Trailing Blank Lines

ReportWriter consolidates trailing and leading blank lines of adjacent report areas. The number of blank lines actually output equals the number of trailing blanks in the first of the two areas or the number of leading blank lines in the second area, whichever is greater.

Suppose, for example, that a page header ends with two blank lines and that the next area to appear on that page is a break header that begins with one blank line. ReportWriter outputs only two blank lines between the text of the page header and the text of the break header.

ReportWriter's pagination procedure accounts for blank line consolidation in computing the amount of space remaining on a page.

NOTE: A blank line contains no display text, fields, or array elements of any kind, whether populated or not. A line consisting of a field that is currently blank is not a blank line in this context; such a line would be affected by the `shrink` keyword (described in the previous section) but does not affect consolidation of leading and trailing blank lines.

Section 10.5.3 suggests a technique for using the `non-display` attribute to suppress the consolidation of blank lines.

6.6
QUERIES

The `query` clause of the `detail` statement is used to specify the data to be retrieved from the database or other input source.

The only processing ReportWriter performs on the query string is colon expansion and, if JAM/DB*i* release 5 is initialized, colon–plus expansion. Otherwise, processing of the query is entirely the responsibility of the program that will retrieve the data.

The query, then, must conform to the requirements of your input processor, whether that be JAM/DB*i* initialized for a particular database or a developer–written row–supply function. The examples in this manual are written in SQL but do not necessarily conform to every one of the SQL dialects used by JAM/DB*i*–supported databases. Some databases, such as Sybase, allow for the use of stored procedures; you can invoke a stored

procedure if supported in your environment. If you are using a custom input function, the query strings must be in the form expected by that function.

While the `query` clause must obey ReportWriter syntax regarding the use of quotation marks and continuation characters, the ReportWriter imposes no requirements on the format and content of the actual query string and performs no processing other than colon (and colon–plus) expansion. Refer to the documentation for the input processor (**JAM**/DB*i* or your custom input function) to determine the correct format of the query string for your application.

6.7
NAMED CURSORS

ReportWriter now permits the use of a named cursor as an alternative to the `query` clause in the `detail` statement.

You must declare the cursor in a JPL or C routine that will be executed before the corresponding `detail` statement is encountered. Execute the cursor by using a `cursor` clause instead of a `query` in the `detail` statement. The syntax of this clause is described in Section 5.1.2.

Named cursors are particularly effective in promoting runtime efficiency when used in subreports that are invoked repeatedly. The cursor is declared just once (typically at startup of the application or of the parent report). The `detail` statement in the subreport then operates by the equivalent of a `DBMS EXECUTE`, which is faster than performing the implicit cursor declaration of the `query` clause each time the `detail` statement is invoked.

Refer to the **JAM**/DB*i* documentation for information on declaring and executing named cursors.

In the following example, a named cursor is declared in a JPL procedure invoked from the primary report. The cursor is executed in the `detail` statement of the subreport.

```
# << begin report >>
#
# init jpl = define_cursor
#
# detail query = "select unique dept, id from employees"
#        report = "vita :dept :id"
#
# << end report >>
#

# << begin report = vita >>
#
# /* The parameters department and emp_id must exist in the LDB or
#    as fields on the report format screen: */
#
# init parameter = department
#      parameter = emp_id
#
# detail cursor = "jobs department, emp_id"
#        area = joblist
#
# << end report >>

# This JPL procedure is invoked from the primary report. It declares
# the cursor that is invoked in the detail statement of the subreport.
# By declaring the cursor only once, no matter how many times the
# subreport is called, the report will run more efficiently.

proc define_cursor
dbms declare jobs cursor for select dept, id from history where \
     dept = ::d and id = ::i order by dept

# The use of d and i as placeholding variables in the preceding cursor
# declaration is standard DBi usage. The names are arbitrary.
```

6.7.1

Reserved Cursor Names

ReportWriter uses named cursors internally to implement the subreport feature. To avoid conflicts with cursor names that report developers are likely to use, ReportWriter reserves the following naming convention for its own use:

`_RWcursor4$`

where:

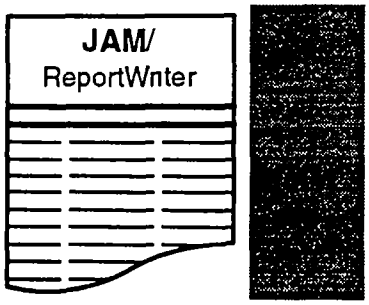
$	is the name of the report whose detail query is being executed

6.7.2

Using the Default Cursor

ReportWriter 5.1, when linked with JAM/DB*i* release 5 does not use the default (unnamed) cursor. Thus, developers can now use JAM/DB*i* `sql` statements in procedures invoked from within the report without disrupting the current query.

If you are using ReportWriter linked with JAM/DB*i* release 4, you can use the `cursor` clause, described above, in lieu of a `query` in the `detail` statement. This frees the default cursor for you to use in any associated procedures.



Chapter 7
Processing Flow

Chapter 5 describes, in detail, how each of the six statements in the script language operates. In doing so, it brings up topics related to the order of processing during report generation.

This chapter presents a consolidation of the order of processing issues introduced in Chapter 5 and further elaborates on the interactions among clauses within a statement and among statements within a script. A generic example is developed here to illustrate these issues.

7.1
ORDER OF SCRIPT STATEMENTS

7.1.1
Invoking Actions Directly

The `detail`, `init`, and `insert` statements all invoke various actions—initiating processing or outputting a report area. Consequently, these statements must appear in the script in the order you want these actions to occur.

To begin the generic example: Suppose you are designing a report that will consist of a title page, followed by a detail report based on data fetched from the database, and concluding with a trailer page that indicates the end of the report.

You will start by initializing report variables and, if running ReportWriter with the stand-alone utility, opening a connection to the database before beginning any of the detail pro-

cessing. Assuming that you've chosen to call these routines from an `init` statement, your script would look something like:

```
# << begin report >>
#
# init   jpl    = init_var
#        jpl    = open_db_conn
#
# insert area   = title-pg
#
# insert newpage
#
# detail query = "SELECT ..."
#        area   = dtl-area
#
# insert newpage
#
# insert area   = trailer
#        jpl    = close_db_conn
#
# << end report >>
```

The `insert newpage` statements force the different report areas to each begin on a new page. Otherwise, the detail report might start on the title page or the trailer on the last detail page.

7.1.2

Page Specifications

Next, suppose that you want headers and footers on the pages in the detail area but not on the title and trailer pages.

The `page` statement does not initiate action as do the three statements discussed above. Rather, it defines what should happen when a specific event—a page break—occurs. Its position in the script is, however, significant, since it affects only those statements that follow it and not those that precede it. `page` affects all statements after it until a `clear` or `clear pagespecs` or another `page` statement is encountered.

Similarly, `clear` (and its variants) does not cause an action, but does affect the processing and output of the statements after it.

To add page headers and footers to the detail area of the report, you would add `page` and `clear` statements to the script, as:

```
# << begin report >>
#
# init   jpl    = init_var
#        jpl    = open_db_conn
#
# insert area   = title-pg
#
# page   header   area   = phead
#        footer   area   = pfoot
#
# insert newpage
#
# detail query = "SELECT ..."
#        area   = dtl-area
#
# clear pagespecs
#
# insert newpage
#
# insert area   = trailer
#        jpl    = close_db-conn
#
# << end report >>
```

Note that `insert newpage` statements follow the `page` and `clear` statements. This is necessary to ensure that the next output invoked will be on a new page governed by the new specifications.

7.1.3

Defining Break Groups

The next step is to add break processing to the example script.

`break` statements can appear anywhere before or after the `detail` statements to which they apply, as long as there is no `clear` or `clear breakspecs` between the `detail` and the `break` statements. The order of `break` statements among themselves does matter, as they form a break hierarchy, the first `break` encountered being the highest level.

`break` statements do not have to be contiguous, but this is recommended practice so that the break hierarchy can be readily identified. For the sake of clarity, it is also recom-

mended that `break` statements be placed immediately before the corresponding `detail` statement.

To continue the example, suppose you want to generate subtotals for each account reported. In addition to adding the appropriate `break` statement to the script, you would need to specify in the `detail query` that the rows be fetched from the database in order of account number so that all transactions related to the same account will appear together and be included in the break group subtotals.

You will also need to invoke a procedure from within the `detail` statement to add the values from the current row to the running totals that will be printed in the break footer. The `break` statement must also invoke a procedure to clear break group totals at the start of each new group.

```
# << begin report >>
#
# init   jpl    = init_var
#        jpl    = open_db_conn
#
# insert area   = title-pg
#
# page   header   area   = phead
#        footer   area   = pfoot
#
# insert newpage
#
# break  field = acct_id
#        header   jpl    = clear_break_tots
#        footer   area   = acct-total
#
# detail query = "SELECT * FROM trans \
#                 ORDER BY acct_id"
#        area   = dtl-area
#        jpl    = do_break_tots
#
# clear pagespecs
#
# insert newpage
#
# insert area   = trailer
#        jpl    = close_db_conn
#
# << end report >>
```

Suppose, too, that you need to report the aggregate totals by state as well as by account. Create a break hierarchy of account within state and reflect this hierarchy in both the order of the `break` statements and in the `detail query`:

```
# << begin report >>
#
# init   jpl    = init_var
#        jpl    = open_db_conn
#
# insert area   = title-pg
#
# page   header   area  = phead
#        footer   area  = pfoot
#
# insert newpage
#
# break  field = state
#        header   jpl   = clear_state_tots
#        footer   area  = state-total
#
# break  field = acct_id
#        header   jpl   = clear_acct_tots
#        footer   area  = acct-total
#
# detail query = "SELECT * FROM trans, acc \
#                 WHERE trans.acct_id=acc.acct_id\
#                 ORDER BY acc.state, trans.acct_id"
#        area   = dtl-area
#        jpl    = do_break_tots
#
# clear pagespecs
#
# insert newpage
#
# insert area   = trailer
#        jpl    = close_db_conn
#
# << end report >>
```

Additional topics related to break processing are covered in Section 7.2.

7.2
ORDER OF CLAUSES

The effects of incorrect placement of clauses and subclauses within script statements can be difficult to correct unless you understand the processing flow among these clauses and their interactions across statements.

7.2.1
Order–sensitive Clauses

In any statement or clause where `area`, `jpl`, and `call` clauses can be used, they are executed in order of appearance. In the `detail` statement, the `breakcheck` keyword is also executed at the point where it is encountered (assuming that it is placed before the `area` clause).

Certain clauses are not sensitive to their order within the statement. For example, `header` and `footer` clauses can be placed anywhere within a `page` or `break` statement without affecting the function of the statement. Thus,

```
#
# page   header   area  = phead
#        footer   area  = pfoot
#
```

will produce exactly the same results as

```
#
# page   footer   area  = pfoot
#        header   area  = phead
#
```

On the other hand,

```
#
# page   footer   jpl   = calc_pnum
#                 area  = pfoot
#
```

may produce different results from

```
#
# page    footer    area   = pfoot
#                   jpl    = calc_pnum
#
```

Break processing is particularly sensitive to the subtleties of clause placement. This topic is covered in the next section.

7.2.2

Multiple Areas and Subreports per Statement

In ReportWriter 5.1, you can output more than one report area and/or subreport in a single statement. Areas can be output with procedure invocations in any combination. For example,

```
# insert    area    =   a1
#           jpl     =   j
#           area    =   a2
```

outputs report area `a1`, executes JPL procedure `j`, and then outputs report area `a2` (if procedure `j` returns `0`).

The following example invokes a subreport after each fetch and then executes JPL procedure `j1`, followed by output of areas `a3` and `a4`:

```
# detail    query   =   "..."
#           report  =   sub1
#           jpl     =   j1
#           area    =   a3
#           area    =   a4
```

In the next example, both a subreport and an area are output as part of the page footer. Note the use of the `reservelines` subclause.

```
# page  footer
#           jpl     =   j2
#           report  =   sub2 reservelines = 3
#           area    =   a5
```

Placement of Qualifying Keywords

Because more than one area is now allowed in any statement, certain keywords that, in earlier releases of ReportWriter, could be placed anywhere within the statement must now be placed directly after the particular area to which they apply. These keywords are `shrink`, `split`, `float`, `nodupl`, and `showattop`.

The following examples show correct placement of qualifying keywords:

```
# page  footer  area = a float shrink
#               area = b

# break field   = f
#       header  area = a showattop
#               area = b showattop nodupl shrink
```

The `nodupl` and `showattop` keywords can also be applied to `jpl` and `call` subclauses as well as to `area`. This allows you to ensure that procedures are executed synchronously with a particular area.

Multiple Areas in Page Footers

Multiple areas are permitted in all statements where the `area` clause or subclause can appear. The only restriction occurs when the `float` keyword is used for page footer areas. In that case, all areas with the `float` designation must be output before any non–floating areas. The following example shows how both floating and non–floating areas can be combined in a page footer.

```
# page  footer  area = pf1 float
#               jpl  = j3
#               area = pf2 float
#               jpl  = j4
#               area = pf3
```

Page Breaks between Areas

ReportWriter allows page breaks to occur between areas output in the same statement except in the following case: If multiple areas are output in the `detail` statement, the first set of detail areas in a break group remains intact and with the break headers. Subsequent sets of detail areas in the same break group may be split across pages. (The individual areas, however, will not be split across pages unless the `split` keyword has been specified.)

7.2.3

Break Processing

Break processing occurs during each iteration of the `detail` statement. It consists of determining if the value of any break field has changed since the last iteration and, if so, performing all break footer processing and output for the previous break group and then

performing all break header processing and output for the new group that begins with the row most recently fetched.

For each loop through the `detail` statement, the first step is to fetch from the database. When the first row is returned, all `area`, `jpl`, and `call` clauses and the `breakcheck` keyword are processed in the order in which they are encountered. If the `breakcheck` keyword is present, break processing occurs at that point; if not, break processing takes place immediately before the `area` clause is executed; if neither the `area` clause nor the `breakcheck` keyword is present, break processing takes place after all `jpl` and `call` clauses have been executed.

Depending upon the types of breaks you have defined, you might have some procedures that must be invoked before break checking and processing and some that must be invoked after.

Procedures that determine the value of a computed break field must precede break processing. The age range example shown in Section 5.2.4 includes a procedure of this type.

Procedures that depend upon some action in the break header must be executed after break processing. Suppose, for example, you have an application in which each detail form reports some value as a percentage of the sum for the break group. The sum is fetched by a routine in the break header and is then available to the detail procedure that computes the percentage for the current row. This procedure must follow break processing so that the correct sum is used in calculating the percentage for the first row in the break group.

In the following example, the JPL procedure `get_sum` fetches the sum of the population for all counties in the state. The procedure `do_percent` calculates the county's population as a percentage of the state total previously calculated in the break header. This percentage is then displayed as a field in the report area `county-pop`.

```
# break  field = state
#        header   jpl   = get_sum
#
# detail query = "SELECT * FROM census\
#                 ORDER BY state, county"
#        breakcheck
#        jpl    = do_percent
#        area   = county-pop
```

Procedures that calculate values required for the break footer must also be invoked after break processing. The `detail` statement from the examples in Section 7.1.3 contains such a procedure:

```
#
# detail query = "SELECT * FROM trans, acc \
#                  WHERE trans.acct_id=acc.acct_id\
#                  ORDER BY acc.state, trans.acct_id"
#        area  = dtl-area
#        jpl   = do_break_tots
#
```

The `do_break_tots` procedure updates the running totals for the current row's break group. (Unlike the previous example, this procedure does not calculate a value needed for the detail area; it can, thus, be placed after the `area` statement, and break checking and processing can occur at their default position.)

If this `jpl` statement were executed before break processing occurred, it would add the current row's values to the previous row's break group. This would cause each running total to be missing the values from the first row that belongs to that group and to include the values from the first row that belongs to the next group. By placing this clause after break processing, the values are added to the correct running totals.

A look at one of the `break` statements from the same example explains why this works:

```
#
# break  field = acct_id
#        header   jpl   = clear_acct_tots
#        footer   area  = acct-total
#
```

Notice that break header processing includes calling a procedure that clears the applicable running total. If the break field has not changed from one row to the next, break processing does not take place, so the running total continues to accumulate. If the break field has changed with the current row, the break footer for the previous group is processed. (In this case an area, which presumably shows the running total, is output.) Then header processing for the new group takes place. Thus, the account totals are cleared out after the previous break group's totals are output but before the current row's values are added, which is just as it should be for the first row in the group.

There is not necessarily one right way to organize a report script. In the `break` statement shown above, the same effect could be achieved by putting the `jpl` subclause in the `footer` clause and omitting the `header` clause altogether:

```
#
# break  field = acct_id
#        footer   area  = acct-total
#                 jpl   = clear_acct_tots
#
```

In this version of the `break` statement, it is necessary to remember that the `area` and the `jpl` subclauses will be executed in the order they are encountered. Since the JPL procedure clears out totals that you want to appear in the footer area, make sure that you output the area first and then clear the totals, as shown above.

A summary of break processing appears in Section 7.2.5.

7.2.4
Computed Breaks

The break examples given so far in this chapter have been based on fields fetched from the database. You can also define breaks based on values generated in JPL procedures or C routines.

Computed breaks can be of many different types, each requiring a slightly different technique to implement:

- inserting special processing or an area after every n^{th} detail line (perhaps an area consisting of just a blank line to improve readability of the report)
- outputting a special area whenever data in the fetched row meet certain criteria (such as an invoice value greater than a specified amount)
- grouping fetched rows into categories derived from the data in the row (such as grouping dated transactions into year quarters)

An example of how to implement this type of break can be found in Section 5.2.4.

The same order–of–processing issues that apply to the breaks described above also apply to computed breaks. In addition, you must keep in mind how the routine that generates the break field value will interact with break processing.

7.2.5
Break Processing Summary

- Break checking occurs during each pass through the `detail` statement (at the point indicated by `breakcheck`, before the `area` clause, or after all `jpl` and `call` clauses).
- If checking reveals that a break field has changed, break processing takes place.

- After the first fetch from the database (but before detail output), all break headers are processed (as if a break had been detected, but, of course, no footer processing is done because there was no prior break group).
- After the last row has been processed, all break footers are processed for the last break group.
- During break header processing, all `area`, `jpl`, and `call` subclauses within the `header` clause are processed in the order they are encountered; colon–expanded variables can be used as arguments to any of these subclauses, as applicable. Break footers are processed in like fashion.
- When a break is detected, that level and all lower–level breaks are processed.
- Multi–level break header processing is from highest–level to lowest.
- Multi–level break footer processing is from lowest–level to highest.

No break or detail processing or output occurs if no rows match the detail query. ReportWriter issues a warning in this case. (Warning messages can be suppressed by using the `-i` option when ReportWriter is invoked; refer to Sections 9.1 and 9.4 for more information on this option.)

7.3
ORDER OF INCLUDED SCREENS

If your report script imports another report format screen that also contains `<<include screen>>` compiler directives, you will want to be aware of the order in which the screens are appended to the primary report format screen. This is an issue only when multiple imported screens contain areas, reports, or procedures bearing identical names.

Each `<<include screen>>` compiler directive in the primary report format screen is processed to its full depth before the next screen is imported.

Suppose, for example, the primary report format screen contains the following `<<include screen>>` compiler directives:

```
<< include screen = x >>
<< include screen = y >>
```

and screen x contains

```
<< include screen = z >>
```

The resulting merged screen would begin with the entire primary report format screen, followed by screen x, screen z, and, finally, screen y.

If an area name tag on an included screen matches one on the primary screen, that area of the included screen is appended to the corresponding area of the main screen. If the same area name tag appears on multiple included screens, they are appended to the primary screen in the order the `<<include screen>>` compiler directives are processed.

To continue the example, suppose the primary report format screen, screen y, and screen z all contain areas with the tag a. The resulting area a on the merged screen would consist of all the a lines from the primary screen, followed by those from screen z and then from screen y.

If you are developing reports that import screens containing `<<include screen>>` compiler directives, you should keep in mind the order in which these directives are processed and the screens appended to the primary screen.

7.4
PAGINATION

ReportWriter applies a variety of rules in determining when it should begin a new page. This section reviews ReportWriter's pagination defaults and explains how to use report script commands to force a page break or to override those defaults you do not want applied.

7.4.1
Keeping Report Areas Intact

By default, ReportWriter will not unnecessarily split a report area across pages.

If the next area to be output will fit in the space remaining on the current page, it is output immediately. If it will not fit on the current page but will fit, in its entirety, on a single page, ReportWriter begins a new page before the area is output.

If the area is longer than a single page, output begins on the current page and continues to the top of the next page. As much of the area as fits on the current page is written, a page break occurs (with its associated processing and footer and header output, if any), and output of the area continues on the next page.

To suppress this feature for any report area (except page headers and footers), add the `split` keyword to the corresponding `area` clause or subclause. When the area is out-

put, it begins on the current page, continuing to subsequent pages, as necessary. Note, however, that the first line of a multi–line area is never orphaned on the bottom of a page, even if the `split` keyword has been specified.

Page headers and footers will never be split over two or more pages. If a page header or footer is too big to fit on a single page, an error occurs, and execution of the report is aborted.

The `split` keyword is also used to override ReportWriter's orphan suppression feature, as described in the Section 7.4.3.

7.4.2
White Space Consolidation

ReportWriter consolidates trailing and leading blank lines of adjacent report areas. The number of blank lines actually output equals the number of trailing blanks in the first of the two areas or the number of leading blank lines in the second area, whichever is greater.

Refer to Section 6.5.2 for a more thorough explanation of this feature. A technique for suppressing blank line consolidation is provided in Section 10.5.3.

7.4.3
Orphan Suppression

By default, ReportWriter requires that all break headers plus at least one instance of detail remain together and unbroken.

When a break occurs, if the current page does not have enough space to accommodate all of the headers that must be output plus at least one instance of the detail area, a new page is begun before the first header is output.

To override the orphan suppression feature, add the `split` keyword to one or more `break header area` subclauses and/or to the `detail area` clause, as appropriate.

Any break header area with the `split` keyword is exempt from the orphan suppression rule and may end, span, or begin a page. If there are multiple headers, this may result in some appearing at the bottom of one page and the remainder at the top of the next.

If `split` is applied to the detail area, not only can the area be split across pages, as described in Section 7.4.1, but it can begin a new page, even if its headers end the previous page.

7.4.4

Effect of Dynamic Report Areas on Orphan Suppression

As described in Sections 6.1.1 and 6.5.1, report areas may be selected and sized dynamically. This flexibility on the part of ReportWriter can, however, present difficulty in determining the amount of space required to effect suppression of orphan break headers. This section explains how and when the computation of required space is performed and shows how the use of dynamic areas can affect its accuracy.

ReportWriter computes the number of lines needed to output all break headers plus one instance of the detail area. The computation takes into account the consolidation of leading and trailing blank lines of adjacent report areas.

This pre-computation takes place after break checking but before break header processing. Therefore, these computations may not reflect the state of the break header or detail area when either is actually output.

Break header JPL or other routines called before header area output might alter that area. Routines processed at any break level can also alter any lower-level break header area, particularly if area selection is dynamic.

Similarly, the detail area can be affected by break header processing or by detail routines invoked after break processing but before area output.

As a result, when a dynamic area is actually output, it might might be a different size from that anticipated when the calculations were performed. This could result in an area or a sequence of headers being split on output, or it might mean that a new page is started to accommodate areas that could have fit on the previous page.

If you are using dynamic report areas and a possible miscalculation of the space required to avoid orphan headers would be a problem, you might try some of the following to minimize the likelihood of this occurring:

- If the detail area is specified dynamically, make sure the routine that fills in the relevant variable is invoked before, rather than after, break processing.
- If break header areas are dynamic, you might want to create them so that all areas that might be used for a given header are the same length.

This way, even if the selected area changes, the computation is unaffected.

- If it is not possible to define all areas for the same header to be the same length, you might want to execute a routine during lowest-level footer processing that places the name of the longest possible area into the variable for each dynamically selected header. This would ensure that the calculated space requirement would be adequate to fit all scenarios.
- If orphan break headers are not considered a problem for your reporting needs, use the `split` keyword.

Similar problems can occur when dynamic page footers are used.

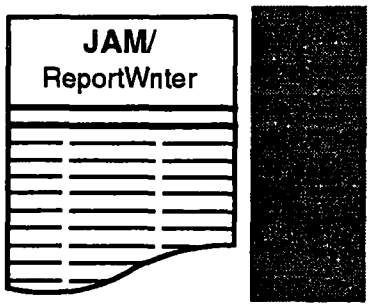
7.4.5

Changing Page Specifications

When a `page`, `clear`, or `clear pagespecs` statement is encountered in the report script, the new page header and footer specifications take effect at the beginning of the next page.

A change in page specifications does not force a page break; if the current page is partially filled, ReportWriter will continue to output to this page either until it is full or until a new page is otherwise forced—by orphan suppression, non-splitting of areas, or the `newpage` keyword. The current page, having been started under the previous page specifications, is closed with the previous page footer.

If you want to close out the currently open page so that the next area output will be on a page with the new header and/or footer specifications, place an `insert newpage` statement either immediately before or immediately after the `page` statement. This will force the current page, if one is open, to end. The next output to occur will begin on a new page. An example is shown in Section 7.1.2.



Chapter 8
ReportWriter Input and Output

8.1
DEVICE CONFIGURATION FILES

Output device support is implemented through the use of device configuration files. Any or all of the following information can be entered into these files:

- initialization and reset strings,
- page width and length,
- left margin,
- output device, file name, or process to which output is spooled, and
- name of the developer–written output function and the size of its output buffer.

The applicable device configuration file, if any, is specified when the JAM/ReportWriter is invoked.

Device configuration files are created as ASCII source files which must be compiled using the `dev2bin` utility.

8.1.1

Format

Each device configuration file is created as an ASCII source file which can contain any of the following statements:

```
init = init_string
reset = reset_string
spool = spool_cmd
procedure = outproc
obuffsize = bufsize
lines = nlines
columns = ncols
leftmargin = nblanks
feedlines = nflines
fixedlength
varlength
```

init_string and ***reset_string*** are the initialization and reset strings for the output device. These strings follow the conventions of JAM video file capability strings: the capabilities are given byte–by–byte, separated by spaces. Non–printable bytes and the space character are represented by their respective ASCII names: NUL, NL, ESC, SP, etc. Any byte can, instead, be represented by its octal value, in the form

```
\ddd
```

where ***d*** is an octal digit. Refer to the chapter on video files in the JAM *Configuration Guide* for further information on the format of initialization and reset strings.

If an `init` string is specified, ReportWriter prefaces the report with the string. If a `reset` string is specified, ReportWriter appends it to the report.

spool_cmd is the name of a program or file. On UNIX systems, output is piped to the specified program. On other operating systems, the output is written to the specified file. The `spool` statement is just one of several ways of directing ReportWriter output; refer to Section 8.2 for information on resolving conflicting output specifications.

outproc is the name of a user-written procedure which writes or filters output. Refer to Section 8.4 for information on writing output functions.

bufsize is the size of the output buffer, in bytes, used by the function ***outproc***. If not specified, the default size is 256 bytes.

nlines specifies the length of the printing area on the report page. ***ncols*** specifies the width of the printing area on the report page. These parameters can also be specified in the report script `init` statement; if they appear in both places, the values in the `init` statement take precedence. If they are not specified either in the `init` statement or in the device configuration file, the page size defaults to 60 lines by 132 columns.

nblanks specifies the number of blank spaces to be prepended to each non-blank line. These spaces must be included in the line length ***ncols***. This parameter can also be specified in the report script `init` statement; if it appears in both places, the value in the `init` statement take precedence. If `leftmargin` is not specified in either location, ***nblanks*** defaults to 0.

nflines is the number of line feed characters that should be used to separate pages. If specified, ***nflines*** plus ***nlines*** must equal the physical length of the page. The `feedlines` parameter can also be specified in the report script `init` statement; if it appears in both places, the value in the `init` statement take precedence. If not specified in either place, or if the value of ***nflines*** is 0, ReportWriter outputs a form feed to begin the next page.

If the `fixedlength` keyword is specified, all report lines are padded with spaces to equal the number of columns specified. If `fixedlength` is not specified, ReportWriter outputs variable-length lines.

If the `varlength` keyword is specified, ReportWriter outputs variable length lines. Since varible length is the default, this keyword is normally not needed in an `init` statement, except to override the `fixedlength` keyword in the device file.

The `fixedlength` and `varlength` keywords are mutually exclusive.

`fixedlength` or `varlength` can also be specified in the report script `init` statement. The keyword specified in the `init` statement overrides whichever of these appears in the device file.

Create the device configuration file with any ASCII editor.

8.1.2
Example

The following is an example of a device configuration file:

```
init = ESC P
reset = ESC Q
lines = 55
columns = 80
spool = lp
```

8.1.3
Compiling the Device Configuration File

The device configuration file must be compiled with the `dev2bin` utility before it can be used by ReportWriter. The invocation sequence from the command line is

`dev2bin [-e` ***ext***`]` ***devfile***

where ***devfile*** is the name of the device configuration file to be compiled. If ***devfile*** does not include an extension, `dev2bin` looks first for ***devfile***`.dev`. If that file cannot be opened, it attempts to open ***devfile*** (with no extension).

The output of `dev2bin` is a binary file named ***devfile.ext***. If the `-e` option is omitted, the resulting file is named ***devfile***`.bin`.

The resulting binary device file can be specified as the argument to the `-d` option of `rwrun`, the report generation utility.

To use a compiled device file in producing a report, invoke ReportWriter with the `-d` option:

`rwrun` ***report*** `-d` ***devfile***

8.2 RESOLVING CONFLICTING OUTPUT SPECIFICATIONS

8.2.1 Destinations

Because ReportWriter provides several different ways to indicate the output destination, the potential exists for conflicting output specifications at runtime. This section explains how ReportWriter chooses the output destination if multiple specifications are present.

When ReportWriter is invoked from the command line, the precedence of output specifications is

1. `procedure` statement in the specified device configuration file
2. output file specified in the command line `-o` option
3. `spool` statement in the specified device configuration file
4. standard output—default if no output destination is otherwise specified

If ReportWriter is invoked from within a JAM/DB*i* application, the command line options are taken from `RWOPTIONS`, which can be either an LDB variable or an environment variable. If both exist, the LDB variable is used. The precedence of output specifications is

1. `procedure` statement in the specified device configuration file
2. output file specified in the `RWOPTIONS -o` option
3. output file specified in the `rwrun` JPL command or in the string passed to the `dbi_rwrun ()` function
4. `spool` statement in the specified device configuration file

Note that there is no default output destination when ReportWriter is invoked from within a JAM/DB*i* application. At least one of the above output specifications must be present; otherwise, an error message is generated.

Refer to Section 9.4 for further information on `RWOPTIONS`.

If the output procedure specified in the device configuration file (precedence #1 in both

lists, above) is invoked and returns with a value of 1, ReportWriter continues to the next output option.

NOTE for users of Microsoft Windows: If no output destination is specified when ReportWriter 5.1 is running under Microsoft Windows, ReportWriter sends its output to the Windows Print Manager. This applies whether ReportWriter is invoked as a stand-alone application or from within a C or JPL routine.

8.2.2
Page Specifications

Some of ReportWriter's page parameters can be specified either in the report script `init` statement or in the device configuration file. In the event of a conflict between parameters, those in the `init` statement take precedence.

It is recommended that you use the `init` statement for parameters that must remain constant no matter where report output is directed. Use the device configuration file for parameters, such as page size, that might vary, depending upon the output device or file.

8.3
DEVELOPER–WRITTEN ROW–SUPPLY FUNCTIONS

ReportWriter uses JAM/DB*i* to perform fetches from the database. You may, however, need to design a report that requires data from flat files, wire services, or other sources not supported by JAM/DB*i*.

This section provides guidelines for writing custom row–supply input functions for a JAM/ReportWriter application.

Instructions for installing your input function and linking it with ReportWriter are provided in Section 8.5.

NOTE: When a user–written row–supply function is installed, ReportWriter neither fetches data nor places it into the report format screen. These tasks become the responsibility of the row–supply function.

8.3.1

The Query

ReportWriter passes the query string in the report script to the row–supply function, having first performed colon expansion, if needed. The format of the query depends upon the format your function expects. It need not be a SQL statement unless, of course, your function is designed to parse a SQL statement.

8.3.2

Arguments

The row–supply function should accept two arguments: an integer type code and a pointer to a string, in that order.

The type code argument is one of

```
RW_P_OPEN
RW_P_CLOSE
RW_P_READ
```

These are defined in the `rwdefs.h` file, included in the ReportWriter distribution. ReportWriter calls the row–supply function once with `RW_P_OPEN` and once with `RW_P_CLOSE`. Between these two calls, ReportWriter calls the row–supply function repeatedly with `RW_P_READ` until there are no more rows to fetch.

When the row–supply function is called with `RW_P_OPEN`, the second argument is the colon–expanded query from the report script. For all subsequent calls, the second argument is the null pointer.

8.3.3

Return Values

The row–supply function should return one of the following:

`RW_USEROK`

on open: The file was successfully opened.

on input: Data were fetched successfully.

on close: The file was successfully closed.

RW_USEREND

on open: Skip the rest of the current `detail` statement and continue with the remainder of the report. This would occur, for example, if the query were the null string.

on input: End of file was reached.

on close: The file could not be closed, but ReportWriter should not abort.

RW_USERABORT

Abort the remainder of the report. The report is aborted, but ReportWriter does not display any specific error message explaining why the report failed. You can use the row–supply function to put out a message, if appropriate.

These return codes are defined in the `rwdefs.h` file, included in the ReportWriter distribution.

8.3.4

Invoking the Row–Supply Function

In addition to installing and linking the input function as described in Section 8.5, you must make the following modification to the `rwopts.c` module supplied with the ReportWriter distribution:

In the `rw_options()` function, set the variable `rw_input_procedure` to the name of the row–supply function. If this variable is set to the null value, ReportWriter uses JAM/DB*i* to fetch data; if it is set to the name of a developer–written procedure, ReportWriter uses that procedure for input.

Alternatively, you may choose to modify `rwopts.c` to allow the end user to choose one of several installed input functions. To do this, define, in `rwopts.c`, a new command line option through which the user can specify the desired input function.

8.4 DEVELOPER–WRITTEN OUTPUT PROCEDURES

This section provides guidelines for writing custom procedures to write or filter ReportWriter output.

To use an installed output function, specify it in the device file `procedure` statement. Use the `obuffsize` statement to specify a buffer size greater than the default of 256 bytes. (Refer to Section 8.1for more information on device configuration files.)

Instructions for installing and linking developer–written output functions are provided in Section 8.5.

8.4.1 Arguments

The output function should accept two arguments: an integer type code and an output buffer, in that order.

The type code argument is one of

```
RW_P_OPEN
RW_P_CLOSE
RW_P_WRITE
```

These are defined in the `rwdefs.h` file, included in the ReportWriter distribution. ReportWriter calls the function once with `RW_P_OPEN`, once per line of output with `RW_P_WRITE`, and once with `RW_P_CLOSE`.

The second argument is an output buffer.

- When the type code is `RW_P_WRITE`, the buffer contains a line of output, terminated by the NEWLINE and NULL characters. Your procedure can modify the contents of the buffer.
- When the type code is `RW_P_OPEN`, the buffer contains the initialization string, as specified in the device file; the string is terminated by the NULL character.
- When the type code is `RW_P_CLOSE`, the buffer contains the reset string, as specified in the device file; the string is terminated by the NULL character.

NOTE: Since the initialization and reset strings are terminated with the NULL character when passed to the output function, neither can contain this character as part of the string.

8 4.2
Return Values

The output procedure should return one of the following:

0	ReportWriter should not output the buffer; the developer–written procedure has handled this step.
1	ReportWriter should output the buffer; the developer–written procedure has only filtered or analyzed the data.
-1	Request for ReportWriter to abort the rest of the report.

8.4.3
Invoking the Output Procedure

To use a custom output procedure with ReportWriter, specify the procedure's name in the `procedure` parameter of the device configuration file. The default size for the output buffer, described in Section 8.4.1is 256 bytes. You can specify a different size through the `obuffsize` parameter of the device configuration file.

Refer to Section 8.1 for more information on device configuration files.

8.5
INSTALLING DEVELOPER–WRITTEN FUNCTIONS

The installation instructions in this section apply to both row–supply functions and developer–written output functions.

1. Install the procedures into the prototyped function list of `funclist.c`. Instructions for this step are provided in the JAM *Programmer's Guide* and in the `funclist.c` file, which is part of the JAM distribution. For each, use the prototype `(i,s)`.
2. Compile `funclist.c`.

3. In `rwmain.c`, make sure that the line

   ```
   sm_do_uinstalls ()
   ```

   is not commented out.

4. In the supplied makefile, append the names of the object files for your custom input and output procedure onto the line that begins

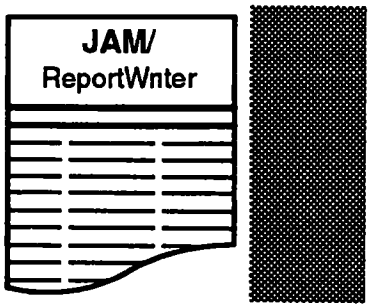
   ```
   USERMODS = funclist.o
   ```

   This step is documented in the makefile. (The exact name of this file depends upon your operating system.)

5. Run the makefile to create the desired executables. This step recompiles `rwmain.c` and relinks `rwrun`, `jamrw`, and/or `jxrw`, depending upon your other modifications to the file. (Refer to instructions in the makefile for specifying which executables to create.)

NOTE: If you are developing input or output functions in C, remember to include the ReportWriter constants described earlier in this chapter:

```
#include "rwdefs.h"
```



Chapter 9
Running ReportWriter

JAM/ReportWriter can be invoked directly from the command line or from within a JAM or JAM/DB*i* application. This chapter explains how to run ReportWriter from

- the command line, using the `rwrun` utility
- a JPL module, using the `rwrun` JPL command
- a C routine, using the `dbi_rwrun ()` function

9.1
FROM THE COMMAND LINE

Use the `rwrun` utility to generate a report directly from the operating system command line. The invocation sequence is

```
rwrun [-d devicefile] [-a|-f] [-i] [-o outputfile] report [arg1 arg2 ...]
```

report must identify a binary report file. It is not necessary to specify the `.bin` extension, as `rwrun` looks first for a file named ***report***`.bin`. If that file cannot be opened, it attempts to open ***report*** (with no extension). The opened file must be the output of the report compiler `rprt2bin`.

arg1, ***arg2***, etc. are the arguments passed to the report. Each argument accepted by the report must be defined by a `parameter` clause in the `init` statement of the report. Refer to Section 6.3 for more information on report arguments.

The `rwrun` options are

`-d` ***devicefile***

Use the parameters in the named device file to control report output. If any parameters specified in the device file are also specified in an `init` statement in the report script, those in the `init` statement take precedence. Refer to Section 8.1 for a detailed description of the device file.

`-a` Append the output to the file named in the `-o` option. If the file does not exist, create it as a new file. This option does not apply if output is sent to a spool command or to a developer–written output procedure.

The final page of each report is closed so that any subsequent report appended will begin on a new page. Note that this behavior of the `-a` option applies only when ReportWriter is invoked as a stand–alone application with the `rwrun` utility.

Refer to Section 9.4 for an explanation of how this option works when ReportWriter is invoked from within a JAM/DB*i* application.

`-f` Allow ReportWriter to overwrite the report output file. This option does not apply if output is sent to a spool command or to a developer–written output procedure.

`-i` Ignore warning messages. If this option is omitted, warning messages are displayed on the screen or sent to the standard destination for error messages in your configuration.

`-o` ***outputfile***

Direct the finished report to the named file. If this option is omitted, and if no spool command or output procedure is specified in the device file, the report is sent to standard output. Refer to Chapter 8 for a more thorough discussion of report output.

The `-a` and `-f` options conflict and should not be specified simultaneously.

9.1.1
Examples

The following examples assume that the report screen `emphist.jam` has been compiled by the `rprt2bin` utility and that the binary file produced is `emphist.bin`.

1. To generate the report and display the output on the terminal screen (standard output), enter the following at the command line:

```
rwrun emphist
```

2. Suppose you are accumulating the output of several reports in a single file, named `emp_reports.txt`, to be printed at a later time. To send the output of `emphist` to this file without overwriting any other reports previously generated and stored in the file, enter:

   ```
   rwrun emphist -a -o emp_reports.txt
   ```

3. You want to send the output of `emphist` to the printer described in the device file `printfile`. Suppose, also, that you expect some warning messages, which you do not want displayed. Enter the following on the command line:

   ```
   rwrun emphist -d printfile -i
   ```

9.2
FROM A JPL PROCEDURE

Use the `rwrun` JPL command to invoke ReportWriter from within a JAM/DB*i* application. The syntax of this JPL command is

`rwrun` ***reportname*** [***outputfile***]

reportname identifies a binary report file. It is not necessary to specify the `.bin` extension, as `rwrun` looks first for a file named ***reportname***`.bin`. If that file cannot be opened, it attempts to open ***reportname*** (with no extension). The opened file must be the output of the report compiler `rprt2bin`.

Output options are specified in the `RWOPTIONS` variable, described in Section 9.4. Alternatively, the output file name can be specified in the `rwrun` command. If ***outputfile*** is specified in the `rwrun` command and the `-o` option is included in the `RWOPTIONS` variable, the file specified in `RWOPTIONS` takes precedence.

Refer to Section 8.2 for further information on resolving conflicting output specifications.

Report arguments are also specified in `RWOPTIONS`. Refer to Section 6.3 for more information on report arguments.

9.3 FROM A C ROUTINE

User-written C functions linked with `jamrw` can invoke ReportWriter with the `dbi_rwrun` library function:

```
err = dbi_rwrun (s)
char *s;
int err;
```

The variable ***s*** points to a string whose contents are the name of a report binary file and, optionally, an output file. If both file names are present, they must be separated by one or more spaces. The format of the string ***s*** is analogous to the argument list for the `rwrun` JPL command:

reportname [***outputfile***]

reportname identifies a binary report file. It is not necessary to specify the `.bin` extension, as `dbi_rwrun` looks first for a file named ***reportname***`.bin`. If that file cannot be opened, it attempts to open ***reportname*** (with no extension). The opened file must be the output of the report compiler `rprt2bin`.

Output options are specified in the `RWOPTIONS` variable, described in Section 9.4. Alternatively, the output file name can be specified in the argument string supplied to the `dbi_rwrun` function. If ***outputfile*** is specified in the argument string and the `-o` option is included in the `RWOPTIONS` variable, the file specified in `RWOPTIONS` takes precedence.

Refer to Section 8.2 for further information on resolving conflicting output specifications.

Report arguments are also specified in `RWOPTIONS`. Refer to Section 6.3 for more information on report arguments.

The return value of `dbi_rwrun` is `-1` if an error occurred; otherwise, it is zero. If, however, the JPL statement `dbms error_continue` is active, `dbi_rwrun` will always return zero.

9.4
RWOPTIONS

When ReportWriter is invoked with the `rwrun` utility, as described in Section 9.1, output options are entered directly on the command line. When ReportWriter is invoked with the JPL `rwrun` command or with the C function `dbi_rwrun ()`, however, any required options must be specified in the variable `RWOPTIONS`.

`RWOPTIONS` can be either an LDB variable or a system environment variable. ReportWriter first looks for `RWOPTIONS` in the LDB. Only if this field is empty or does not exist, does ReportWriter check `RWOPTIONS` in the system environment.

In either case, `RWOPTIONS` is a string containing ReportWriter options. Any of the following options can appear in `RWOPTIONS`:

`-a` Append the output to the file named in the `-o` option. If the file does not exist, create it as a new file. This option does not apply if output is sent to a spool command or to a developer–written output procedure.

The final page of the report is left open so that the next report output will be appended without forcing a page break.

If ReportWriter is invoked successively with `-a` and the same output file specified, unexpected results can occur if the reports involved use different page footers. Refer to Section 9.4.2 for further information on this option.

If both the `-a` and the `-c` options are specified, the `-c` option takes precedence.

Note that the `-a` option works somewhat differently when ReportWriter is invoked with the stand–alone utility `rwrun`. Refer to Section 9.1 for an explanation of how this option works in that case.

`-c` Close the last page of the latest report appended to the file named in the `-o` option. No new report is generated nor is any other output produced.

If ReportWriter is invoked first with `-a` and then with `-c`, unexpected results can occur if the reports involved use different page footers. Refer to Section 9.4.2 for further information on these options.

This option does not apply if output is sent to a spool command or to a developer–written output procedure.

This option is ignored when ReportWriter is invoked from the stand–alone utility `rwrun`.

`-d` ***devicefile***
: Use the parameters in the named device file to control report output. If any parameters specified in the device file are also specified in an `init` statement in the report script, those in the `init` statement take precedence. Refer to Section 8.1 for a detailed description of the device file.

`-f`
: Allow ReportWriter to overwrite the report output file. This option does not apply if output is sent to a spool command or to a developer–written output procedure.

`-i`
: Ignore warning messages.

`-o` ***outputfile***
: Direct the finished report to the named file. If this option is omitted, and if no spool command or output procedure is specified in the device file, the report is sent to standard output. Refer to Chapter 8 for a more thorough discussion of report output.

The `-a`, `-c`, and `-f` options conflict and should not be specified at the same time.

9.4.1

Format

Options are entered into `RWOPTIONS` in the same format they would be entered on the command line.

The following example shows how the LDB variable `RWOPTIONS` might be setup and ReportWriter invoked from JPL.

```
cat RWOPTIONS "-a -d mydevice -o myoutput"
rwrun myreport
```

Note that the output file can also be specified as an argument to `rwrun`. The following is equivalent to the preceding example.

```
cat RWOPTIONS "-a -d mydevice"
rwrun myreport myoutput
```

If an output file is specified in the `rwrun` command and also in `RWOPTIONS`, output is sent to the file named in `RWOPTIONS`. In the following example, ReportWriter will send the output to `myoutput`.

```
cat RWOPTIONS "-a -o myoutput"
rwrun myreport yourfile
```

Refer to Section 8.2 for a more a more complete discussion of conflicting output specifications.

9.4.2

Append and Close Options: -a and -c

The append option, -a, when invoked from within a JAM/DB*i* application, causes output to be appended to the previous report in the file without forcing a page break between reports. The close option, -c, closes the final page of the last report appended to the output file; it produces no additional output.

The -a option is used primarily when ReportWriter is invoked in a loop, producing individual reports that are essentially similar parts of a larger report driven by the JAM/DB*i* application. This is a super–query, as described in Section 10.1, "Sub–Reports."

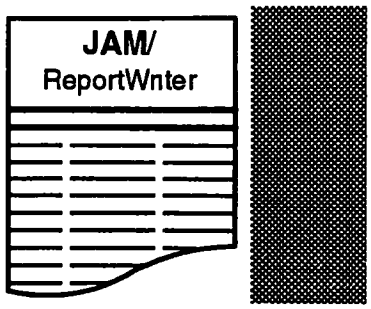
The following example shows how a JAM/DB*i* application might invoke ReportWriter in a loop to produce a continuous (no page breaks between invocations) section of a report. It subsequently invokes ReportWriter again to close the final page of the continuous report section so that the next report appended to the output file will begin on a new page. Finally, it invokes ReportWriter again to create another report appended to the first.

```
dbms count dbcount
cat pagenum "0"
system rm -f outfile
cat RWOPTIONS "-a"

dbms declare c cursor for SELECT ...
dbms execute c

while dbcount > 0
{
   rwrun report_1 outfile
   dbms continue
}

cat RWOPTIONS "-c"
rwrun report_1 outfile

cat RWOPTIONS "-a"
```

```
rwrun report_2 outfile

cat RWOPTIONS "-c"
rwrun report_2 outfile
```

If ReportWriter is invoked in a loop and you want each iteration of the report to begin on a new page, place an `insert newpage` statement at the beginning of the script.

The reports specified by the `rwrun` commands that are executed with the `-a` and `-c` options need not be the same. Unexpected results can arise, however, if the two reports use different headers and footers. In general, you will want to close the last output page by specifying the same report that was used to generate the output, as shown in the example above.



Chapter 10
Development Hints

10.1
ALTERNATIVE METHOD FOR SUBREPORTS

The subreport capability described in Section 6.2 of this manual applies only when ReportWriter is linked with

- JAM release 5.03a or higher and
- JAM/DB*i* release 5.

If you are using an earlier version of JAM or JAM/DB*i*, you may want to use the techniques shown here as an alternate means of producing subreports.

When linked with releases of JAM, and JAM/DB*i* earlier than those noted above, ReportWriter uses the unnamed (default) cursor of JAM/DB*i*. Thus, it can have only one query open at a time. Notice, for example, that while multiple `detail` statements are permitted in a report script, they are processed sequentially rather than being nested or interleaved. This does not, however, preclude the possibility of creating subreports. JPL procedures or other developer–written functions, whether invoking or invoked from ReportWriter, can declare and use named cursors, which will not conflict with the unnamed cursor used in the detail query. Two techniques are available for nesting queries to create subreports:

- Sub–Queries: The ReportWriter `detail` statement provides the outer query, and a JPL procedure (or called function) fetches additional data with a named cursor.

- Super–Queries: The JAM/DB*i* application performs the main query, using a named cursor, and invokes ReportWriter in a loop to fetch lower–level data as needed.

The following example uses both of these techniques to create a multi–level report. (Section D.3.2 of this manual shows how this example would be implemented using the full subreport capability of ReportWriter release 5.1.)

This report lists individuals and account balances. For any with a negative balance, a subreport is produced showing the creditors and the amount owed to each. A further level of detail is produced wherever the individual owes $200 or more to a particular creditor. Figure 13 shows a section of the sample report.

```
Mary                                        +400
Paul                                        -400
      Butcher                       -100
      Baker                         -200
                  DETAIL
                  ------
            donuts          50
            cupcakes        30
            bread           20
            turnovers       30
            b'day cakes     70
      Candlestick Maker           -100
Peter                                       +150
```

Figure 13: A Report With Subreports

The top–level query, which fetches information about individuals, is performed in a JPL module of the JAM/DB*i* application. ReportWriter is invoked in a loop to fetch the creditor information for each individual reported and, in turn, to invoke a JPL procedure to fetch further creditor detail information.

ReportWriter handles the report output, including the data fetched in the main application. The data must, of course, be in the LDB for ReportWriter to have access to it.

Since the creditor subreport and purchase detail are not required in all cases, this report area is populated only as needed. The `shrink` keyword associated with the area in the report script eliminates excess white space. (Refer to Section 6.5.1 for further information on sizing reports dynamically.)

Figure 14 shows a portion of JPL code from the main application. It performs the top-level query and invokes ReportWriter in a loop to fetch subsequent levels of data. The

```
.
.
.
dbms count dbcount

# append ReportWriter output to the file
# "output"

cat RWOPTIONS "-a -o output -i"

dbms declare c cursor for select person, \
     sum (bal) bal from all_bal \
     group by person
dbms execute c

# for each person fetched from the database,
# run the report oneperson:

while dbcount > 0
{
      rwrun oneperson
      dbms continue c

}

# close the final page of the oneperson report:

cat RWOPTIONS "-c -o output"
rwrun oneperson
.
.
.
```

Figure 14: The Super–Query: Invoking ReportWriter in a Loop

invoked report is named `oneperson`. Figure 13 shows three iterations of `oneperson`. The append output option (`-a`) is used so that they form a single report.

```
# << begin report>>
#
# /* This is the "oneperson" report: */
#
# insert jpl      = format_individual_bal
.
.
.
# insert area   = individual
#
# detail query = "select person, vendor, \
#                  bal balance \
#                  from all_bal \
#                  where person = ':person' \
#                  and bal < 0"
#         jpl    = purchases
#         area   = creditors shrink
#         jpl    = reset_creditors_flds
.
.
.
# << end report>>
```

Figure 15: Script for Report `oneperson`

Figure 15 shows portions of the `oneperson` report, including the `detail` statement. This statement

- performs the database query to fetch the creditor information for the individual selected in the main query and
- invokes the JPL procedure `purchases`.

The `purchases` procedure, shown in Figure 16, is invoked during each iteration of the `detail` statement. The SQL `select` statement in this procedure is the sub–query to fetch further details on the individual's account with the creditor.

```
proc purchases

# get the list of purchases only if person
# owes $200 or more:

if balance <= -200
{
   # "item" and "item_price" are arrays:

   dbms declare d cursor for select item, \
               item_price from stores \
               where buyer = ':person' and\
                     store = ':vendor'
   dbms execute d

   # Put title on this section of report only if
   # it will be used. Otherwise, leave this field
   # blank so the entire subreport can be shrunk
   # out if not needed.

   if dbcount >0
   {
      cat item_label "DETAIL"
      cat underscore "------"
   }
}
```

Figure 16: The Sub–Query: Using Detail JPL to Fetch Further Data

```
person__________                    bal_____   <<individual
                                               <<individual

      vendor______       balance____           <<creditors
           item_label__                         <<creditors
           underscore__                         <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
      item_____    item_price__                 <<creditors
```

Figure 17: The Report Areas (with Field Names)

The layout of the report screen is shown in Figure 17. The report area `individual` is output once each time ReportWriter is invoked from the application code. The fields in this area, `person` and `bal`, are filled in from the main query.

The area `creditor` is output once for each line fetched by the query in the `detail` statement. It may, however, be empty or only partially filled, depending upon the individual's balance. The `shrink` keyword is, therefore, used in the script where this area is output.

The `vendor` and `balance` fields are filled from the `detail` query, the `item` and `item_price` arrays from the sub–query (in the `purchase` procedure). `item_label` and `underscore`, which form the title for the purchase detail section, are filled in only if the sub–query returns data.

10.2
GIVING THE END USER CONTROL OVER REPORT COMPOSITION

Although JAM/ReportWriter reports are developed in the authoring environment rather than at runtime, end users can still be given a considerable amount of control over report content and composition.

This section shows how to use colon–expanded variables in the report script in conjunction with various user interface and coding techniques to make report composition sensitive to the user's input. (Refer to Section 6.1.1 for more information on colon–expanded variables.)

Design the user interface of your application to solicit relevant information about the report from the user. Use data entry fields, menus, checklists and/or radio buttons, as appropriate, to allow the user to make the report data and content selections.

The information may apply to the data to be fetched, such as:

```
Enter the year for which you want benefits data: _____
```

Or it may specify which format, or variation, of a report area to use:

```
Which employee history format do you want?

          _ brief        _ full
```

In the first of the above examples suppose that the data entry field is named `year`. The `query` clause of the `detail` statement might then be something like:

```
# detail   query   = "SELECT * from emp_benefits \
#                     WHERE yr = :year"
```

In the second example, you might have two alternate areas defined. One might be called `emphist-brief` and the other `emphist-full`. The `area` clause of the `detail` statement might look something like:

```
# detail   query   = "..."
#          area    = :selection
```

Your application code would determine the user's selection from this radio button group and insert the applicable area name into the variable `selection`. Suppose the radio button group name is `button`. Your application JPL might contain the following:

```
if button == 1
    cat selection "emphist-brief"
else
    cat selection "emphist-full"
```

You can also use colon-expanded variables to include or exclude various processing or area output. For example:

```
Do you want totals by salesperson? (y/n) _
```

In this example, your report script might invoke, as part of break footer processing, JPL to calculate the running total for each salesperson. After the detail area is output, a separate report area listing those totals is output. You want this processing and output to occur only if the user requests it. The `break` and `insert` statements might look like:

```
# break     field   = acctnum
#           footer  jpl     = :calc
#                   area    = bfoot
.
.
.
# insert    area    = :totals
```

The JPL code in the application is responsible both for capturing the user's response to the query about reporting the totals and for assigning appropriate values to the variables `calc` and `totals`. Suppose the input variable in the prompt above is `pquery`; the application JPL might contain:

```
if pquery
{
    cat calc "calc_salesperson"
    cat totals "tot_area"
}
else
{
    cat calc ""
    cat totals ""
}
```

Note that the application code assigns the null string to these variables if the user does not want the totals computed and reported. When the value of an argument to an `area`, `jpl`, or `call` clause is the null string, ReportWriter simply bypasses the clause without causing an error and without attempting to output an area or invoke any code.

10.3
REPORTS DEVELOPED UNDER REPORTWRITER RELEASE 4

Reports developed under ReportWriter release 4 can be run under release 5 if they are first converted with the `rw4to5` utility.

`rw4to5` converts reports developed under JAM/ReportWriter release 4 into release 5 format. This utility takes as input the report script file, the JAM screens, and the JPL modules that make up the release 4 report and produces as output a single report format screen that can be compiled by the release 5 `rprt2bin` utility.

The `rw4to5` utility is described in Chapter 13, the Utilities Reference.

If your report, under ReportWriter release 4, made use of the LDB for passing the values of report variables, you may be able to use non–output fields on the release 5 report format screen instead. Under ReportWriter release 5, the report format screen is open throughout report generation, so the content of all screen fields is always accessible. This makes it possible (and, indeed, preferable) to use non–output fields on the screen in place of LDB variables. Refer to Section 6.1.2 for more information about the scope of ReportWriter variables and the use of non–output fields.

10.4
RUNNING RELEASE 5.0 REPORTS UNDER RELEASE 5.1

All ReportWriter release 5.0 report format screens and device configuration files must be recompiled with the release 5.1 `rprt2bin` and `dev2bin` utilities, respectively.

The source code, however, need not be changed.

10.5
INTERACTIONS WITH JAM FEATURES

ReportWriter is closely integrated with JAM and derives much of its functionality from the base product. A few of JAM's screen composition and navigation features, however, are not applicable to report development and are ignored when used with report format screens. This section provides guidelines to help you take maximum advantage of ReportWriter's integration with JAM.

10 5.1

Fields and Arrays

Word–Wrapped Arrays

Word–wrapped arrays work as they do in JAM/DB*i*: If a target variable for the `SELECT` is a word–wrapped array, a single row is fetched on each cycle through the `detail` statement, and text in long columns is apportioned into as many elements of the word–wrapped destination as are needed.

Shifting and Scrolling

Scrolling arrays are meaningless in reports and should not be used. Similarly, shifted fields are irrelevant in report generation. ReportWriter outputs only the onscreen portions of fields and arrays.

Onscreen Arrays

JAM/DB*i* permits the use of arrays as target variables for database fetches. As noted above, however, any such arrays should be onscreen arrays in the detail area. Each cycle through detail processing will fetch as many rows as the elements in the detail area will accommodate.

Bear in mind, however, that fetching into arrays differs from fetching into single–element target variables. ReportWriter invokes procedures specified in `jpl` and `call` clauses on each cycle through the query. When the target variables are simple fields, this means that these procedures are executed for each row fetched. When the target variables are arrays, these procedures are executed once for each fetch, not for each row.

Similarly, break checking and processing occur once for each cycle through the query. To implement break checking and processing for each row, the data must be fetched into single–element fields. When data is fetched into arrays of, say, ***n*** elements, only the first of the ***n*** rows retrieved is checked for breaks.

Thus, the nominal case of detail processing, as described in Section 5.1, occurs when data is fetched into single–element fields.

You may, however, encounter circumstances where it is appropriate to fetch into onscreen arrays. Refer to Appendix B, Section B.2, for additional information on using arrays as target variables with JAM/DB*i*.

10.5.2
Screen and Field Functions

In general, you should avoid attaching functions to the report format screen or to fields on the screen.

Instead, invoke screen entry and exit actions and field validation procedures explicitly through JPL and CALL clauses appropriately placed in the report script. This simplifies debugging of the report by making all procedure and function calls visible in the script, rather than having some of the processing take place "behind the scenes."

The above recommendation notwithstanding, it is useful to know how JAM/ReportWriter interacts with the various screen and field functions, if present. Such might be the case, for example, if you were developing a report format screen from a JAM application screen that had associated screen and/or field functions.

Screen Entry and Exit Functions

If a screen entry function is specified for the report format screen. JAM executes it automatically when ReportWriter opens the screen (before the script is executed). Similarly, the screen exit function, if one is specified, is executed when the report is closed, after the script has been executed.

As noted above, use of these functions is not recommended.

Field Functions

Field Validation Function

Field validation is performed prior to output in order to apply field and miscellaneous edits (refer to Section 10.5.4). Therefore, if a validation function is specified for the field, it, too, is executed.

If custom edits or other validation are required, invoke the needed procedures directly through the script rather than in a field validation function.

If any field fails JAM field validation, an unrecoverable runtime error occurs. If, however, your "validation function" is invoked through the script, your code can control the resulting behavior through appropriate use of return values (described in Section 6.4.2).

Field Entry and Exit Functions

Field entry and exit functions have no meaning in report generation and are ignored.

10.5.3

Display Characteristics

Most display attributes, such as blinking, highlight, underline, etc., specified for fields and text on the report format screen are ignored.

Non–Display

The `non-display` attribute, however, is observed. If a field or piece of text has this attribute, it is not output.

This behavior is useful if you want to apply the `shrink` feature to the report area but need to protect a particular line from being shrunk out, even if all fields on it are empty. Place some text or a field on the line and give it the `non-display` attribute. If all other fields on the line are blank, the line will appear as a blank line in the report, the presence of the non–display text making it immune from the effect of `shrink`.

The `non-display` attribute can be used in a similar manner to prevent consolidation of leading and trailing blank lines. Place a field or a printable character on either the last trailing blank line of the first area or first leading blank line of the second area; give this field or display text the `non-display` attribute. On output, the line will be blank, and ReportWriter will not attempt to suppress it.

10.5.4

Field and Miscellaneous Edits

Most field and miscellaneous edits are irrelevant in the context of report generation and are ignored. The following field and miscellaneous edits, however, do apply to fields on the report format screen:

```
right justified
null field
currency format
date or time field
```

ReportWriter gets the system date and time when the report format screen is opened. This date and/or time is entered into any field designated as a `system date/time` field.

To update the contents of a system date/time field prior to output, invoke a procedure that clears the field, such as:

```
proc update_sys_time
     cat rprt_time
```

The current system date and/or time will be entered into the cleared field.

10.5.5
Borders and Line Drawing

Borders and line drawing do not convert well to printed output, and their use is not recommended.

If these features are used on the report format screen, their graphics characters are converted to various text characters, depending on the border or drawing style chosen. On screens incorporated with the `include screen` compiler directive, borders are omitted.

10.5.6
Colon Preprocessing

Colon–expanded JAM variables are permitted within any user–defined character string in the report script. JAM/DB*i* colon–plus preprocessing may be used in the query string if the string is passed through JAM/DB*i* release 5. The following are legal ReportWriter statements:

```
# break    field  = :var1
#          header area = ":var2"

# detail   query  = "SELECT ... WHERE x = :+var3"
```

Colon–expanded JAM variables are not permitted where a ReportWriter reserved word, a file name, or a numeric value is expected. The following are not legal ReportWriter statements:

```
# init lines = :var1

# :arbitrary_report_script_statement

# clear :what_to_clear

# <<include screen = :filename>>
```

Refer to Section 6.1.1 for a further discussion of colon–expanded JAM variables.

10.5.7

Screen Manager Functions

When ReportWriter is invoked from the command line with the `rwrun` utility, functions that apply only to the keyboard (such as `sm_input`) or to the physical screen (such as `sm_rescreen`) are not supported; data access routines (such as `sm_putfield` and `sm_getfield`) are.

As a general rule-of-thumb, the following types of functions are supported in the stand-alone ReportWriter:

- functions that provide access to
  - field and array data
  - field and array attributes
  - groups
  - the LDB
  - global data
- mass storage and retrieval functions
- functions that manipulate JAM's behavior (except those functions that involve the keyboard)
- miscellaneous functions, such as `sm_l_close`, `sm_l_open`, `sm_jplload`, and `sm_sdtime` (except those functions that involve the keyboard)
- message display functions (behave differently than they do in JAM—display results to standard error output without pausing for user acknowledgement)

When invoked from within a JAM or JAM/DB*i* application, ReportWriter supports all Screen Manager functions. The behavior of these functions is the same as it is in JAM.

10.5.8

Control Strings

Control strings associated with the report format screen are ignored.

10.5.9
Math Precision and Formatting

Precision specifications in JPL `math` statements are a convenient means of on–the–fly formatting, especially for numeric fields, such as page numbers, that have no currency edit. For example, the statement

```
math %.0 page = page + 1
```

overrides the `math` statement's default precision of two decimal places.

10.5.10
Screen Editing and Documentation Facilities

When you are creating the report format screen, remember that you are working in the JAM Screen Editor and have all its capabilities available to you. For example, when modifying the existing contents of a screen, you can use not only the Screen Editor's Move, Copy, and Repeat Last Action function keys, but also the INSERT LINE, DELETE LINE, and other JAM logical keys.

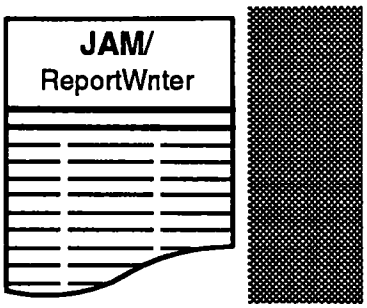
Similarly, JAM's `lstform` utility is especially recommended to help you document the report.

10.6
INTERACTIONS WITH JAM/DB*i*

Programs invoked from the ReportWriter script can use the JPL statements `sql` and `dbms` (or their C language equivalents) to take advantage of all JAM/DB*i* features, with the following exceptions:

- Do not use the default (unnamed) cursor in any program invoked from a `detail`, `break`, or `page` statement, as ReportWriter uses the unnamed cursor to effect the detail query. All database access from such programs should be performed using the `dbms declare cursor` and `dbms execute` statements, rather than the `sql` statement.
- ReportWriter inhibits JAM/DB*i*'s error processing. While ReportWriter is active, JAM/DB*i* uses its default error handling, regardless of the

use of `dbms error` statements. Do not execute `dbms error` from programs invoked via ReportWriter `jpl` or `call` clauses.



Chapter 11
Script Statement Reference

This chapter contains a reference page for each report script statement. The statements are listed alphabetically, and each entry includes

- a synopsis of the statement, including a listing of available keywords and arguments,
- a description of the statement's operation, and
- an example illustrating the statement's use.

The typographical conventions used here are listed in Section 1.4 of this manual.

For a more thorough description of the operation of each statement, its interactions with other script statements, and more extensive examples, refer to Chapter 5 of this manual.

The six ReportWriter script statements are:

`break:`	define a break field and its action
`clear:`	cancel page and/or break specifications
`detail:`	specify the action for each row fetched from the database
`init:`	initialize the report
`insert:`	output an area, invoke a subreport, and/or execute one or more procedures
`page:`	specify page headers and footers

A Reminder about Arguments to Script Clauses:

Wherever the `field`, `query`, `cursor`, `jpl`, `call`, `report`, or `area` clause or subclause can appear, the argument following the equal sign must be one of the following:

- a string of letters, digits, hyphens, and/or underscores, e.g.,

```
# break field = emp_no
```

- a colon–expanded JAM variable, e.g.,

```
# page footer area = :which_area
```

- any string, possibly including colon–expanded JAM variables, enclosed in quotation marks, e.g.,

```
# detail query = "SELECT * from emp_table \
#                 WHERE emp_no = ':empid'"
```

break

define break field and action

SYNOPSIS

```
break field = fieldname [(start[, length])]
      [header  {[area = head-area
                  [shrink][split]
                  [nodupl] [showattop]]* |
               [report  =  "h_invocation_string"
                  [preserve]
                  [preserve breakspecs]
                  [preserve initspecs]
                  [preserve pagespecs]
                  [reservelines = h_number]]* |
               [jpl = hjpl]* |
               [call = hfunction]*}]
      [footer  {[area = foot-area
                  [shrink][split][nodupl]]*|
               [report  =  "f_invocation_string"
                  [preserve]
                  [preserve breakspecs]
                  [preserve initspecs]
                  [preserve pagespecs]
                  [reservelines = f_number]]* |
               [jpl = fjpl]* |
               [call = ffunction]*}
               [noorphanbreak [lines = nlines]]
      [norepeat] [norepeatattop] [newpage]
```

DESCRIPTION

The `break` statement defines a break field and the actions to occur when the value of the break field changes. Multiple `break` statements can be used to define a hierarchy of data breaks; the first defines the highest level break, the last, the lowest.

fieldname is the field on which this break occurs. It must be a field on the report format screen or an LDB variable. Optionally, the break field can be a substring of the specified field, starting at position ***start***, for a length of ***length*** characters. Array elements can also be used as break fields, as explained in Section 5.2.5.

head-area is the name of a report area to be printed in the header for each break group.

h_invocation_string specifies a subreport to be generated as part of the header for each break group; if arguments are passed to the subreport through the invocation string, the entire string must be enclosed in quotation marks. ***h_number*** specifies the number of lines the subreport will occupy.

hjpl is the name of a JPL procedure to be invoked during break header processing.

hfunction is the name of a C (or other supported language) function to be invoked during break header processing.

Any number of `area`, `report`, `jpl`, and `call` subclauses can appear in the `header` clause; they are executed in the order they are encountered.

If the `nodupl` keyword is specified for a header area, that area is not output if the next higher-level break output occurs at the same time. All other header subclauses (except for other `area` subclauses with this keyword) are executed regardless of whether or not a higher-level break has occurred.

The `showattop` keyword directs ReportWriter to output the specified header area at the top of each page, after the page header, whether or not a break has occurred at that point. The `showattop` keyword does not affect any other header subclauses; `report`, `jpl`, `call`, and `area` subclauses without this keyword are executed only when the field has broken.

foot-area is the name of a report area to be printed in the footer for each break group.

f_invocation_string specifies a subreport to be generated as part of the footer for each break group; if arguments are passed to the subreport through the invocation string, the entire string must be enclosed in quotation marks. ***f_number*** specifies the number of lines the subreport will occupy.

fjpl is the name of a JPL procedure to be invoked during break footer processing.

ffunction is the name of a C (or other supported language) function to be invoked during break footer processing.

Any number of `area`, `report`, `jpl`, and `call` subclauses can appear in the `footer` clause; they are executed in the order they are encountered.

If the `nodupl` keyword is specified for a footer area, that area is not output if the next higher-level break occurs at the same time. All other footer subclauses (except for other `area` subclauses with this keyword) are executed regardless of whether or not a higher-level break has occurred.

If the `noorphanbreak` keyword is present and if the group just printed had only one member, the footer areas and subreports are not printed. All `jpl` and `call` subclauses

associated with the `footer` clause are executed, however. If the `lines = nlines` clause modifies `noorphanbreak`, the specified number of blank lines is output in lieu of the footer area.

If the `shrink` keyword follows the name of an area, ReportWriter shrinks that area vertically by removing lines that consist solely of empty fields or empty trailing array elements.

If the `split` keyword follows the name of an area, ReportWriter allows that area to begin, end, or span a page. For headers, the default is to keep them all unbroken and together, along with at least one instance of the detail area. The `split` keyword allows the break header to be left as an orphan at the end of a page while subsequent break headers and/or the corresponding detail area are output on the next page. For footers, the default is to begin on a new page if there is not enough room remaining on the current page to accommodate the entire area.

If any of the `preserve` keywords follows a subreport invocation string, the relevant specifications currently in effect for the parent report take precedence over those in the subreport script.

`norepeat` suppresses output of the break field in the detail area except when the value of the field changes or is at the top of a new page. `norepeatattop` is similar to `norepeat` except that the break field is output only when its value changes.

If `newpage` is specified, each break group after the first begins on a new page.

EXAMPLE

See following page.

```
# /* The following break statements define a break
#    hierarchy of genus, species, and subspecies. */
#
# break  field =  genus
#        header   area  = head
#        footer   area  = gfoot
#        newpage  norepeat
#
# break  field =  species
#        footer   area  = sfoot
#                 jpl   = sfoot_proc
#        header   area  = shead
#                 nodupl
#        norepeat
#
# break  field =  subspecies
#        footer   area  = ssfoot
#        header   area  = sshead
#                 nodupl
```

clear

cancel page or break specifications

SYNOPSIS

```
clear [pagespecs] [breakspecs]
```

DESCRIPTION

The `clear` statement cancels all previous page and/or break specifications. This statement is processed as it is encountered in the report script, cancelling specifications that precede it in the script, but not affecting those placed after it.

The keywords `breakspecs` and `pagespecs` can, optionally, be included to indicate that only the break or page specifications are to be cancelled. If neither keyword is present, `clear` cancels all previous `page` and `break` statements.

A `clear` or `clear pagespecs` statement cancels all previous page specifications. Subsequent pages of the report will show no page headers or footers unless another `page` statement follows.

A `clear` or `clear breakspecs` statement cancels all break processing enabled by previous `break` statements. Break specifications must first be cleared before defining a new break hierarchy; otherwise, any new `break` statement would define a break subordinate to those already in effect.

EXAMPLES

See following page.

```
# /* This example clears an existing break hierarchy --
#    in preparation for a new detail statement
#    and its associated break statements --
#    without changing the page headers and
#    footers currently in effect. */
#
# break ...
# break ...
# detail ...
#
# clear breakspecs
#
# break ...
# break ...
# detail ...
#


# /* End a report with an area "trailer" on a page with
#    no headers or footers */
#
# clear pagespecs
#
# insert newpage
#
# insert area = trailer
```

detail

specify action for each row fetched from the database

SYNOPSIS

```
detail {query  = "sql_statement" | cursor  =  "cur_invoc_string" }
       [area = area [shrink][split]]*
       [report  = "rpt_invoc_string"
         [preserve]
         [preserve breakspecs]
         [preserve initspecs]
         [preserve pagespecs]
         [reservelines = number]]*
       [jpl = jpl]*
       [call = function]*
       [breakcheck] [newpage]
```

DESCRIPTION

The `detail` statement specifies the database query for fetching rows from the database and defines the action that should occur as each fetch is made.

When a `detail` statement is encountered in the report script, ReportWriter processes the SQL statement in the `query` clause or executes the named cursor; then, as each fetch is made from the database, it executes the `area`, `report`, `jpl`, and `call` clauses. ReportWriter continues to cycle through the detail statement in this manner until no more rows are fetched from the database.

The `area`, `report`, `jpl`, and `call` clauses and the `breakcheck` keyword are executed in the order they are encountered. `breakcheck` is ignored if it appears after the first `area` or `report` clause.

Each `detail` statement must have exactly one `query` clause or one `cursor` clause. Both may not appear in the same `detail` statement, but one must be present.

sql–statement directs how rows are fetched from the database. It must be enclosed in quotation marks. If break processing is enabled, the order in which rows are fetched must be consistent with the hierarchy of break fields.

If a row–supply (input) function is installed, the query is passed to this function instead of to JAM/DB*i*. In this case, the query statement must conform to the syntax required by the row–supply function.

The SQL statement is often complex and may require multiple lines. The line continuation character is a backslash (\); it must be the last character on each continued line.

cur_invoc_string specifies a named cursor to be invoked to fetch rows from the database; if arguments are passed to the cursor through the invocation string, the entire string must be enclosed in quotation marks. The named cursor must be declared in a JPL or C routine that is invoked before the `detail` statement is encountered.

area is the name of a report area to be output for each row fetched.

rpt_invoc_string specifies a subreport to be generated for each row fetched; if arguments are passed to the subreport through the invocation string, the entire string must be enclosed in quotation marks. ***number*** specifies the number of lines the subreport will occupy.

If any of the `preserve` keywords follows a subreport invocation string, the relevant specifications currently in effect for the parent report take precedence over those in the subreport script.

jpl is the name of a JPL procedure to be invoked during detail processing for the row.

function is the name of a C (or other supported language) function to be invoked during detail processing.

Any number of `area`, `report`, `jpl` and `call` clauses can be specified.

If the `shrink` keyword follows the name of the area to output, ReportWriter shrinks the area vertically by removing lines that consist solely of empty fields or empty trailing array elements.

If the `breakcheck` keyword is specified, break checking and processing take place at the point where this keyword is encountered. Otherwise, break checking and processing take place immediately before the first area or subreport is output or, if no `area` or `report` clauses are present, after all `jpl` and `call` clauses have been executed. The `breakcheck` keyword is effective only if it appears before the first `area` or `report` clause.

The `split` keyword allows a detail area to begin, end, or span a page. The default is to begin an area on a new page if there is not enough room remaining on the current page to accommodate the entire area (as well as any required break headers). The `split` keyword also allows the corresponding break headers to be left as orphans on the previous page.

If `newpage` is specified, a new page is forced at the end of each pass through the detail statement unless no output resulted from that pass. `newpage` has no effect if there is neither an `area` clause nor a subreport that generates output in the `detail` statement.

EXAMPLE

```
# /* For each row fetched from the database,
#    perform the JPL procedure "prebreak" before
#    break checking and processing. After break
#    processing, perform the procedure "postbreak"
#    and output the area "dtlarea." This area has a
#    large array that may not be fully populated;
#    to save space on the printed report, do not
#    print trailing blank lines that result from
#    unused array elements. */
#
# detail query =  "SELECT * FROM accts \
#                  ORDER BY state, acctid"
#        jpl    =  prebreak
#        breakcheck
#        jpl    =  postbreak
#        area   =  dtlarea shrink
```

init

initialize the report

SYNOPSIS

```
init  { [area = area [shrink][split]]* |
        [jpl = jpl]* |
        [call = function]* |
        [newpage] |
        [lines = nlines] |
        [columns = ncols] |
        [leftmargin = nblanks] |
        [feedlines = nflines] |
        [fixedlength] | [varlength] |
        [parameter = arg_name]* }
```

DESCRIPTION

The `init` statement performs report initialization. This is usually the appropriate place to invoke procedures that open a connection to the database (if you are running ReportWriter from the stand–alone utility `rwrun`) and initialize report variables.

In addition, the `init` statement can be used to specify page size, left margin, and other output parameters for the report. These parameters can, alternatively, be specified in a device configuration file. Any output parameter specified in the `init` statement overrides the corresponding parameter in the device file.

Use the `init` statement, also, to define the arguments that can be accepted by this report.

The `area`, `jpl`, and `call` clauses are executed in the order they are encountered.

area is the name of a report area to be output when the `init` statement is executed.

jpl is the name of a JPL procedure to be invoked during during initialization.

function is the name of a C (or other supported language) function to be invoked at this time.

Any number of `area`, `jpl`, and `call` clauses can be specified.

If the `shrink` keyword follows the name of an area to output, ReportWriter shrinks the area vertically by removing lines that consist solely of empty fields or empty trailing array elements.

The `split` keyword allows the area to begin, end, or span a page. The default is to begin an area on a new page if there is not enough room remaining on the current page to accommodate the entire area.

If `newpage` is specified, subsequent output will appear on a new page.

nlines specifies the length of the printing area on the report page. ***ncols*** specifies the width of the printing area on the report page. If these parameters are not specified either in the `init` statement or in the device configuration file, the page size defaults to 60 lines by 132 columns.

nblanks specifies the number of blank spaces to be prepended to each non–blank line. These spaces must be included in the line length. If `leftmargin` is not specified, it defaults to 0.

nflines is the number of line feed characters that should be used to separate pages. If specified, ***nflines*** plus ***nlines*** must equal the physical length of the page. If not specified, or if the value of ***nflines*** is 0, ReportWriter uses a form feed to begin each new page.

If the `fixedlength` keyword is specified, all report lines are padded with spaces to equal the number of columns required by the columns specifier ***ncols***. If `fixedlength` is not specified, ReportWriter outputs variable length lines.

If the `varlength` keyword is specified, ReportWriter outputs variable length lines. Since variable length is the default, this keyword is normally not needed except to override the `fixedlength` keyword in the device file.

The `fixedlength` and `varlength` keywords are mutually exclusive.

Each `parameter` clause specifies an argument to be accepted by this report. ***arg_name*** is the JAM variable to receive the value of the next unprocessed argument in the invocation string or in `RWOPTIONS`. If all arguments have been exhausted, the value of ***arg_name*** remains unchanged. The order of `parameter` clauses determines the order in which arguments must be passed to this report.

The `init` statement can be repeated in a report script. Any new page size parameters specified in subsequent `init` statements override the previous ones.

EXAMPLES

```
# /* This ınit statement invokes JPL routines that
#    initıalize report variables and open a connection
#    to the database. It outputs the area "titlepage"
#    and sets up the page size parameters. Printing will
#    occur in an area 58 lines long by 64 characters
#    wide (72 - 8 = 64). */
#
# init   jpl    = ınit_rpt_var
#        jpl    = open_db_conn
#        area   = titlepg
#        newpage
#        lines       =  58
#        columns     =  72
#        leftmargin  =  8


# /* The report in this example accepts two arguments.
#    The first is used in the WHERE clause of the detail
#    query. The other specifies the report area to use
#    in the page header. */
#
# << begin report >>
#
# init   jpl = startup
#        parameter   =  parm1
#        parameter   =  parm2
#
# page   header   area  =  :parm2
#        footer   jpl   =  pnum
#                 area  =  pfoot
#
# break  field =  custno
.
.
# detail query "SELECT * FROM orders \
#               WHERE sales_id = ':parm1'\
#               ORDER BY cust_no"
.
.
# << end report >>
```

insert

output an area and/or invoke one or more procedures

SYNOPSIS

```
insert   { [area = area [shrink][split]]* |
           [report  =   "rpt_invoc_string"
             [preserve]
             [preserve breakspecs]
             [preserve initspecs]
             [preserve pagespecs]
             [reservelines = number]]* |
           [jpl = jpl]* |
           [call = function]* |
           [newpage] }
```

DESCRIPTION

Use the `insert` statement to output one or more report areas, invoke one or more subreports, and/or to specify procedure(s) to be executed. Any area output, subreport generation, or procedure invocation specified in this statement is performed once only. (Contrast this with the `detail` statement, which is driven by a database query and performs area output and associated processing for each fetch.)

Typical uses for the `insert` statement include

- producing a title page for the report,
- generating a trailer page showing grand totals accumulated during report generation,
- forcing a page break so that the next report area will begin on a new page, or
- invoking a JPL procedure that performs any necessary cleanup at the end of a report, including closing connections to the database.

The `area`, `report`, `jpl`, and `call` clauses are executed in the order they are encountered.

area is the name of a report area to be output.

rpt_invoc_string specifies a subreport to be generated; if arguments are passed to the subreport through the invocation string, the entire string must be enclosed in quotation marks. ***number*** specifies the number of lines the subreport will occupy.

If any of the `preserve` keywords follows a subreport invocation string, the relevant specifications currently in effect for the parent report take precedence over those in the subreport script.

jpl is the name of a JPL procedure to be invoked.

function is the name of a C (or other supported language) function to be invoked.

Any number of `area`, `report`, `jpl`, and `call` clauses can be specified.

If the `shrink` keyword follows the name of an area to output, ReportWriter shrinks the area vertically by removing lines that consist solely of empty fields or empty trailing array elements.

The `split` keyword allows the area to begin, end, or span a page. The default is to begin an area on a new page if there is not enough room remaining on the current page to accommodate the entire area.

The `newpage` keyword causes subsequent output to appear on a new page. If the `insert` statement contains an `area` clause, the page break occurs after the area is output. If no `area` clause is present, `newpage` simply ensures that whatever area is output next will begin on a new page.

EXAMPLE

```
# /* The first insert statement forces a page break,
#    ensuring that the report trailer area will begin
#    on a new page.
#
#    The second invokes processing to produce data for
#    the report trailer page, outputs the area, and
#    invokes a JPL procedure to perform any needed
#    cleanup and close the connection to the database.
# */
#
# insert newpage
#
# insert jpl    =  do_totals
#        area   =  trailer
#        jpl    =  close_db_conn
```

page

specify page headers and/or footers

SYNOPSIS

```
page  { [header   {[area = head-area [shrink]]* |
                  [report  = "h_invocation_string"
                     [preserve]
                     [preserve breakspecs]
                     [preserve initspecs]
                     [reservelines = h_number]]*  |
                  [jpl = hjpl]* |
                  [call = hfunction]*} ] |
        [footer   {[area = foot-area [shrink][float]]* |
                  [report  = "f_invocation_string"
                     [preserve]
                     [preserve breakspecs]
                     [preserve initspecs]
                     reservelines = f_number ]* |
                  [jpl = fjpl]* |
                  [call = ffunction]*}}
```

DESCRIPTION

The `page` statement defines the page headers and/or footers and their associated processing.

head-area is the name of a report area to be printed in the header on each page.

h_invocation_string specifies a subreport to be generated as part of the page header; if arguments are passed to the subreport through the invocation string, the entire string must be enclosed in quotation marks. ***h_number*** specifies the number of lines the subreport will occupy.

hjpl is the name of a JPL procedure to be invoked during page header processing.

hfunction is the name of a C (or other supported language) function to be invoked during page header processing.

Any number of `area`, `report`, `jpl`, and `call` subclauses can appear in the `header` clause; they are executed in the order they are encountered.

foot-area is the name of a report area to be printed in the footer on each page.

f_invocation_string specifies a subreport to be generated as part of the page footer; if arguments are passed to the subreport through the invocation string, the entire string must be enclosed in quotation marks. ***f_number*** specifies the number of lines the subreport will occupy. The `reservelines` subclause is required for any subreport invoked in a page footer.

fjpl is the name of a JPL procedure to be invoked during page footer processing.

ffunction is the name of a C (or other supported language) function to be invoked during page footer processing.

Any number of `area`, `report`, `jpl`, and `call` subclauses can appear in the `footer` clause; they are executed in the order they are encountered.

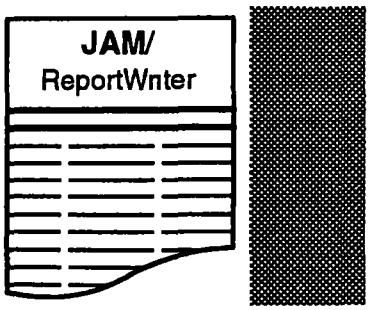
If the `float` keyword appears in an `area` subclause of the `footer` clause, that area immediately follows the last printed line on the page. Otherwise, it appears at the bottom of the page. All areas with the `float` designation must be output before any non-floating areas.

If the `shrink` keyword follows the name of either area, ReportWriter shrinks that area vertically by removing lines that consist solely of empty fields or empty trailing array elements.

`page` can be repeated in a report script.The header specifications in a `page` statement override the header specifications of any previous `page` statements. Similarly, the footer specifications override those in any previous `page` statements. Use `clear` or `clear pagespecs` to cancel out all current page headers and footers.

EXAMPLES

```
# /* The area "head" contains fixed text. The
#    area "foot" contains a field "page."
#
#    The JPL procedure "init_vars" contains the line:
#
#        CAT page "0"
#
#    The JPL procedure "incr_pg" contains the line:
#
#        MATH page = page + 1
#
#    Maintain numbered pages by the following: */
#
# init   jpl    = init_vars
#
# page   header   area  = head
#        footer   jpl   = incr_pg
#                 area  = foot


# /* This example shows how both floating and
#    non-floating areas can be combined in a
#    page footer. */
#
# page   footer   area =   pf1 float
#                 jpl  =   j3
#                 area =   pf2 float
#                 jpl  =   j4
#                 area =   pf3
```



Chapter 12
Library Function Reference

This chapter contains a reference page for each of the library functions supplied with JAM/ReportWriter. Each reference page contains a synopsis of the statement, including a listing of available keywords and arguments, and a description of the statement's operation and return values.

The typographical conventions used here are listed in Section 1.4 of this manual.

The library includes a function for invoking ReportWriter from a JAM/DB*i* application. This function is:

- `dbi_rwrun:` invoke JAM/ReportWriter

It also includes three functions that are called directly from `rwmain.c`, the main routine for `rwrun`. Information on these functions is provided to assist developers who are creating customized versions of the report generation program. These functions are:

- `rw_init:` initialize the report generator
- `rw_options:` parse ReportWriter options
- `rw_run:` produce a report

dbi_rwrun

invoke the report generator from a user–written function

SYNOPSIS

```
err = dbi_rwrun(s);
int err;
char *s;
```

DESCRIPTION

This function is used to invoke the report generator from a user–written function linked into a JAM/ReportWriter application.

The variable ***s*** is a string whose contents are the name of a report binary file and, optionally, an output file. If both file names are present, they must be separated by one or more spaces. The format of the variable ***s*** is analogous to the argument list for the `rwrun` JPL command:

reportname [***outputfile***]

reportname identifies a binary report file. It is not necessary to specify the `.bin` extension, as `dbi_rwrun` looks first for a file named ***reportname***`.bin`. If that file cannot be opened, it attempts to open ***reportname*** (with no extension). The opened file must be the output of the report compiler `rprt2bin`.

Output options are specified in the `RWOPTIONS` variable, described in Section 9.4. Alternatively, the output file name can be specified in the argument string supplied to the `dbi_rwrun` function. If ***outputfile*** is specified in the argument string and the `-o` option is included in the `RWOPTIONS` variable, the file specified in `RWOPTIONS` takes precedence.

Refer to Section 8.2 for further information on resolving conflicting output specifications.

`dbi_rwrun()` is intended for use within a JAM/ReportWriter application, such as `jamrw`. It must be called after the JAM initialization normally performed by `jmain.c` or `jxmain.c`. To invoke ReportWriter from a non–JAM application such as `rwrun`, use, instead, the functions `rw_init` and `rw_run`.

RETURNS

−1 = error
0 = no error

If the JPL statement `dbms error_continue` is active, `dbi_rwrun` always returns zero.

EXAMPLE

The following example uses the syntax for JAM/DB*i* Release 4:

```
sm_n_putfield ("RWOPTIONS", "-f");
dbi_dbms ("error"); /* use default error processing */

if (dbi_rwrun("myreport myoutput") == -1)
{
   sm_n_putfield ("myrwstatus", "failure");
}
```

rw_init

initialize the report generator and the **JAM** screen manager

SYNOPSIS

```
rw_init ();
```

DESCRIPTION

This function is called from the stand–alone ReportWriter utility main program, `rwmain.c`, to initialize the JAM screen manager and the ReportWriter globals. It must be called before any JAM function, or the function `rw_run`, is executed.

EXAMPLE

```
rw_init ();
sm_1_open ("myjp1library");
if (rw_run (report, device, output))
{
   exit (1);
}
```

rw_options

parse ReportWriter options

SYNOPSIS

```
error = rw_options (argc, argv, report, device, output);
int error;
int argc;
char *argv[];
char **report;
char **device;
char **output;
```

DESCRIPTION

This routine parses the options accepted by the `rwrun` utility or used by a JAM/ReportWriter application.

The options are null–terminated strings in ***argv***, with the count in ***argc***. The variables ***report***, ***device***, and ***output*** are set to the relevant file names, as specified by the arguments.

When ReportWriter is invoked as a stand–alone utility, the arguments ***argc*** and ***argv*** are the command line arguments. When ReportWriter is invoked from the `rwrun` JPL command or from the `dbi_rwrun()` library function, the arguments ***argc*** and ***argv*** are constructed from the ***reportname*** and ***outputfile*** arguments in the calling sequence and from the LDB or environment variable `RWOPTIONS`.

The source code for `rw_options` is provided in `rwopts.c`, which is supplied with the ReportWriter distribution.

RETURNS

−1 = error in options
0 = no error

EXAMPLE

See following page.

```
input = (char*) NULL;
device = (char*) NULL;
output = (char*) NULL;

if (rw_options (argc, argv, &input, &device, &output)
    == -1 || input == (char*) NULL)
{
   fprintf (stderr, "Usage error\n");
   exit (1);
}
if (rw_run (input, device, output)== -1)
{
   exit (1);
}
```

rw_run

produce a report

SYNOPSIS

```
error = rw_run (report, device, output);
int error;
char *report;
char *device;
char *output;
```

DESCRIPTION

This routine generates the report specified by the argument ***report***.

report must be a compiled report format screen, the output of the `rprt2bin` utility. ***device*** is the name of the compiled device file, the output of the `dev2bin` utility; and ***output*** is the name of a file to which ReportWriter output is to be sent. The values of these three arguments are normally supplied by `rw_options()`, which is called before `rw_run()`.

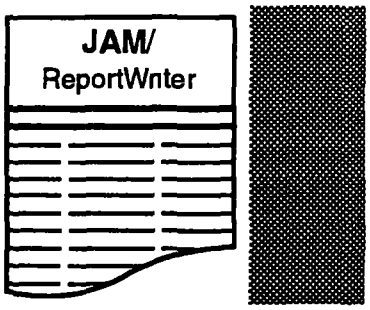
RETURNS

−1 = error
0 = no error

All errors generate messages on the standard error file and are fatal. Examples include errors detected in the binary file, inability to access report areas specified in the binary file, and the absence of a prior call to `rw_init`.

EXAMPLE

```
rw_init ();
if (rw_rwrun ("myreport", (char*) NULL, "myoutput")
    == -1)
{
   exit (1);
}
```



Chapter 13
Utilities Reference

This chapter contains a reference page for each of the utilities supplied with JAM/ReportWriter. Each reference page contains a synopsis of the command, including a listing of available keywords and arguments, and a description of the utility's operation.

The typographical conventions used here are listed in Section 1.4 of this manual.

The following utilities are supplied with ReportWriter:

`dev2bin:`	compile a device configuration source file
`rprt2bin:`	compile a report format screen
`rw4to5:`	convert a ReportWriter 4 report to ReportWriter 5 format
`rwrun:`	run ReportWriter
`tbl2r:`	create a report format screen from a database table

`dev2bin`, `rprt2bin`, and `rw4to5` are provided as executables. The makefile supplied with your ReportWriter distribution makes the stand–alone utility `rwrun`, as well as the interactive JAM/ReportWriter applications `jamrw` and `jxrw`. The `tbl2r` utility must be linked with the JAM/DB*i* release 5 libraries.

You may wish to customize `rwrun`. Source code for `rwmain.c`, the main routine for `rwrun`, is provided for this purpose. Refer to Appendix B, Section B.1, for instructions on customizing ReportWriter.

dev2bin

compile a device configuration file

SYNOPSIS

```
dev2bin [-e ext] filename
```

OPTIONS

-e *ext* specifies the extension for the device binary file; if this option is omitted, the resulting file is named *filename*.bin.

DESCRIPTION

The dev2bin utility produces a device binary file from the device configuration file identified by *filename*. If the name does not include an extension, dev2bin looks first for *filename*.dev. If that file cannot be opened, it attempts to open *filename* (with no extension).

The output of dev2bin is a binary file named *filename*.*ext*. If the -e option is omitted, the resulting file is named *filename*.bin.

The resulting binary device file can be specified as the argument to the -d option of rwrun, the report generation utility, or in the LDB or environment variable RWOPTIONS for use by the interactive program jamrw.

rprt2bin

compile a report format screen

SYNOPSIS

```
rprt2bin [-e ext] rprtname
```

OPTIONS

-e *ext* specifies the extension for the report binary file; if this option is omitted, the resulting file is named *rprtname*.bin.

DESCRIPTION

The rprt2bin utility produces a report binary file from the report format screen identified by *rprtname*. If *rprtname* does not include an extension, the default extension specified in the environment variable SMFEXTENSION is assumed.

The output of rprt2bin is a binary file named *rprtname.ext* If the -e option is omitted, the resulting file is named *rprtname*.bin.

If the report script in the report format screen *rprtname* contains any

```
<< include screen = >>
```

compiler directives, the JAM screens indicated in those directives are compiled and incorporated into the resulting binary file.

The report binary file is used as input to rwrun, the report generation utility, or as an argument to the JPL statement rwrun in JAM/DB*i*.

The report binary file cannot be edited. If you need to change the report in any way after it has been compiled, you must make the changes to the source file for the report format screen and recompile.

Refer to Chapters 4 and 5 for a description of the format and content of the source file to be compiled by the rprt2bin utility.

rw4to5

convert a ReportWriter 4 report to ReportWriter 5 format

SYNOPSIS

`rw4to5` ***script*** `[-f] [-k] [-n] [-v] [-o` ***outputfile***`] [-e` ***ext***`]` ***screen... jpl...***

OPTIONS

`-f` Output file may overwrite an existing file.

`-k` Retain the screen file name extension in area name tags. If this option is omitted, `rw4to5` removes the `.jam` extension from the file name before converting it to a name tag on the ReportWriter 5 report format screen. If the original script uses the `.jam` extension in its `form` clauses, use this option when converting the report.

`-n` Do not add a report script to the output file. A dummy argument, however, is required in place of the ***script*** argument on the command line. Use this option if you are creating a report format screen with areas only, intended for inclusion in other reports.

`-v` Generate a list of files processed.

`-o` ***outputfile***
Direct the output to the named file. If this option is omitted, the output file is created in the same directory as the report script.

`-e` ***ext*** Append the specified extension to the report format screen. By default, `rw4to5` uses the value in the environment variable `SMFEXTENSION`.

DESCRIPTION

This utility creates a ReportWriter release 5 report format screen from a release 4 report script and its associated screen and JPL files.

script is the name of the file containing the report script in ReportWriter release 4 format. The full file name, including extension, of the report script file must be entered; no default extension is assumed.

screen is the name of a JAM screen used as a `form` in the release 4 report script. If no extension is specified, `rw4to5` attempts to open a file with the file name ***screen*** and the default extension specified in the environment variable `SMFEXTENSION`.

jpl is the file name of a JPL module referenced (either directly or by its named procedures) in the release 4 report script. The full file name, including extension, of each JPL module must be entered; no default extension is assumed.

The new report format screen is built on a copy of the first specified screen. The screen JPL, attached JPL, and control strings of this screen are kept, while those of the remaining screens are dropped. The report script is inserted as comments at the top of the screen JPL. Area name tags assigned in the release 5 report format screen are taken from the file names of the corresponding release 4 screens.

The JPL procedures are also added to the screen JPL of the report format screen. Unnamed procedures are given the name of the JPL file from which they are taken. The remaining specified screens are appended to the end of the report format screen.

If the `-o` ***outputfile*** option is not specified, `rw4to5` uses the full ***script*** file name, including extension, and appends the extension specified in the `-e` ***ext*** option. (See the examples below.)

When the converter drops a screen's attachments, a warning message is displayed. All borders are also dropped, and warning messages are displayed.

EXAMPLES

The following examples show how the name of the resulting report format screen is generated:

```
rw4to5 example.rpt exdetail.jam exshead.jpl
```

produces ReportWriter 5 report format screen in the file named `example.rpt.jam`

```
rw4to5 example.rpt -o example2 exdetail.jam exshead.jpl
```

produces ReportWriter 5 report format screen in the file named `example2.jam`

```
rw4to5 example.rpt -o example2 -e form exdetail.jam exshead.jpl
```

produces ReportWriter 5 report format screen in the file named `example2.form`

rwrun

run ReportWriter

SYNOPSIS

```
rwrun [-d devicefile] [-a|-f] [-i] [-o outputfile] report [arg1 arg2 ...]
```

OPTIONS

`-d` ***devicefile***

Use the parameters in the named device file to control report output. If any parameters specified in the device file are also specified in an `init` statement in the report script, those in the `init` statement take precedence. Refer to Section 8.1 for a detailed description of the device file.

`-a` Append the output to the file named in the `-o` option. If the file does not exist, create it as a new file. This option does not apply if output is sent to a spool command or to a developer–written output procedure.

The final page of each report is closed so that any subsequent report appended will begin on a new page. Note that this behavior of the `-a` option applies only when ReportWriter is invoked as a stand–alone application with the `rwrun` utility.

Refer to Section 9.4 for an explanation of how this option works when ReportWriter is invoked from within a JAM/DB*i* application.

`-f` Allow ReportWriter to overwrite the report output file. This option does not apply if output is sent to a spool command or to a developer–written output procedure.

`-i` Ignore warning messages. If this option is omitted, warning messages are displayed on the screen or sent to the standard destination for error messages in your configuration.

`-o` ***outputfile***

Direct the finished report to the named file. If this option is omitted, and if no spool command or output procedure is specified in the device file, the report is sent to standard output. Refer to Chapter 8 for a more thorough discussion of report output.

The `-a` and `-f` options conflict and should not be specified simultaneously.

DESCRIPTION

The report defined by the binary file ***report*** is generated and the output sent to the file named in the `-o` option, if present.

report must identify a binary report file. `rwrun` looks first for a file named ***report***`.bin`. If that file cannot be opened, it attempts to open ***report*** (with no extension). The opened file must be the output of the report compiler `rprt2bin`.

arg1, ***arg2***, etc. are the arguments passed to the report. Each argument accepted by the report must be defined by a `parameter` clause in the `init` statement of the report. Refer to Section 6.3 for more information on report arguments.

tbl2r

create a report format screen from a database table

SYNOPSIS

```
tbl2r table [table...] [-i] [-u user [-p password]] [-s server]
[-d database] [-e ext] [-f] [-l] [-y datdic]
```

OPTIONS

-i Run this utility in interactive mode. This opens a screen on which you can enter any information not specified on the command line.

-u *user* Log on with the given user name.

-p *password*
Log on with the given password.

-s *server*
Log on to the named server.

-d *database*
Log on to the named database.

-e *ext* Append the specified extension to the report format screen. By default, `tbl2r` uses the value in the environment variable `SMFEXTENSION`.

-f Overwrite an existing report format screen file.

-l Use lowercase when creating report format screen file names.

-y *datadic* Use the named data dictionary. If this option is omitted, `dbutildd.dic` is used.

DESCRIPTION

NOTE: This utility must be linked with the JAM/DB*i* release 5 libraries.

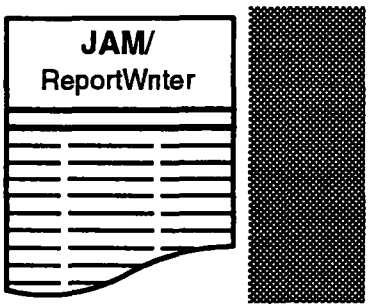
The `tbl2r` utility creates a report format screen for each database table named on the command line. Each report format screen created by `tbl2r` contains

- A field for each column in the table, with up to 250 fields created in total. Field characteristics are assigned according to the column's data type. A field is named for its column in the table.
- Display text on the screen identifying the name of each field.

- A report script template in the screen JPL. This rudimentary script specifies the page header (the display text), the detail area (the fields), and the detail query (`"select * from` ***table***`"`).

This utility is a convenient way to begin a JAM/ReportWriter report. JAM fields are assigned PF4 characteristics compatible with their respective database columns. The area containing the fields is labelled `<<detail`. The area containing the display text that identifies the fields is labelled `<<header`.

The output of `tbl2r` is a JAM screen named ***table.ext***. If the `-e` option is omitted, the resulting file is is given the extension specified in the environment variable `SMFEXTENSION`.



Appendix A
Glossary of Reserved Words

This appendix lists all JAM/ReportWriter reserved words alphabetically, summarizes the purpose of each, and indicates where it can appear in a report script. Refer to the "Script Statement Reference," Chapter 11, or to "Using the Script Statements," Chapter 5, for more detailed and context-oriented information on the use of these reserved words.

area clause or subclause; indicates a report script area to be output; a clause in any of the following statements:

```
detail
init
insert
```

a subclause in the following clauses:

```
break header
break footer
page header
page footer
```

break statement; specifies a break field and the break header and footer processing and output to occur when the value of the break field changes

breakcheck keyword; indicates where break checking and processing should occur within detail processing; used in the `detail` statement

breakspecs keyword; used in the `clear` statement to cancel current break specifications

call clause or subclause; indicates a C (or other supported language) function to be invoked; a clause in any of the following statements:

```
detail
init
insert
```

a subclause in the following clauses:

```
break header
break footer
page header
page footer
```

clear statement; cancels page and/or break specifications

columns clause (output parameter); used in the `init` statement to specify the width of the printing area on the report page; also used in the device configuration file

cursor clause; invokes the specified named cursor to fetch data from the database or other source; used in the `detail` statement

detail statement; specifies the action and output as data are fetched from the database or other input source

feedlines clause (output parameter); used in the `init` statement to specify the number of line feed characters to be used between pages; also used in the device configuration file

field clause; used in the `break` statement to specify the break field

fixedlength keyword (output parameter); used in the `init` statement to specify that report lines must be of fixed length; also used in the device configuration file

float keyword; indicates that page footer should appear immediately after the last report line on the page; used in the `footer` clause of the `page` statement

footer clause; used in the `break` statement to specify processing and/or output to occur at the end of a break group; used in the `page` statement to specify processing and/or output to occur at the end of a page

header clause; used in the `break` statement to specify processing and/or output to occur at the beginning of a break group; used in the `page` statement to specify processing and/or output to occur at the beginning of a page

init statement; initialize the report and specify certain output parameters

insert statement; output an area and/or invoke one or more procedures

jpl clause or subclause; indicates a JPL procedure to be invoked; a clause in any of the following statements:

```
detail
```

```
init
insert
```

a subclause in the following clauses:

```
break header
break footer
page header
page footer
```

leftmargin clause (output parameter); used in the `init` statement to specify the number of columns reserved for the left margin; also used in the device configuration file

lines 1) subclause; used in the `break footer noorphanbreak` subclause to indicate the number of blank lines to be output in lieu of the footer

2) clause (output parameter); used in the `init` statement to specify the length of the printing area on the report page; also used in the device configuration file

newpage keyword; forces subsequent output to begin on a new page; used in the following statements:

```
break
detail
init
insert
```

nodupl keyword; suppresses break header or footer output if the next higher level break or header output occurs at the same time; used in the `break header` and `break footer` clauses

noorphanbreak clause; suppresses break footer output if the break group consists of only one member; used in the `break footer` clause

norepeat keyword; suppresses output of the break field except when its value changes or at the top of a page; used in the `break` statement

norepeatattop keyword; suppresses output of the break field except when its value changes; used in the `break` statement

page statement; specify page headers and/or footers

pagespecs keyword; used in the `clear` statement to cancel current page header and footer specifications

parameter clause; used in the `init` statement to specify an argument accepted by this report

preserve keyword; used in the `report` clause to indicate that initialization, page, and break specifications in the parent report should override the corresponding specifications in the invoked subreport

preserve breakspecs
keyword; used in the `report` clause to indicate that any break specifications in the invoked subreport should be added to the hierarchy established in the parent report

preserve initspecs
keyword; used in the `report` clause to indicate that initialization specifications in the parent report should override the corresponding specifications in the invoked subreport

preserve pagespecs
keyword; used in the `report` clause to indicate that page specifications in the parent report should override the corresponding specifications in the invoked subreport

query clause; directs how data are fetched from the database or other source; used in the `detail` statement

report clause or subclause; invokes the named subreport; a clause in either of the following statements:

```
detail
insert
```

a subclause in the following clauses:

```
break header
break footer
page header
page footer
```

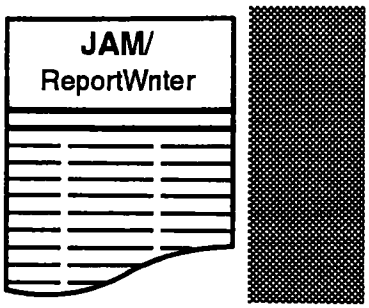
reservelines
subclause; used in the `report` clause to indicate the maximum number of lines the subreport will occupy

showattop keyword; causes the break header to be output at the top of each new page, whether or not a break has occurred at that point; used in the `break header` clause

shrink keyword; shrink the report area vertically to eliminate any rows that consist only of unpopulated trailing array occurrences or blank fields; used with the `area` clause or subclause

split keyword; allows the associated area to begin, end or span a page; overrides ReportWriter's default pagination rule which will not unnecessarily split an area across pages; used with the `area` clause or subclause; when used in the `break header area` subclause, allows the header to be separated from its first detail area or from other headers occurring at the same time

varlength keyword (output parameter); used in the `init` statement to specify that report lines may be variable length; also used in the device configuration file



Appendix B
Implementation Notes

B.1
CUSTOMIZING REPORTWRITER

`rwrun`, as made from the distributed makefile, is capable of producing any report compiled by `rprt2bin`.

Some report developers, however, may need to create a custom version of the report generation program. For example, a custom executable that always generates the same report would eliminate the need for a report name argument on invocation.

For this reason, `rwmain.c`, the main routine for `rwrun`, and `rwopts.c`, the argument–processing module, are included in the JAM/ReportWriter distribution. You can modify or replace either module to create a custom report generator.

`rwmain.c` includes instructions for modifying ReportWriter, as well as for linking in developer–written functions invoked in the report script or through an output procedure specification. The only library functions accessed directly from `rwmain.c` are `rw_init` and `rw_run`. These functions are described in Chapter 12, the Library Function Reference.

You can alter `rwopts.c` if you want to modify the options accepted by ReportWriter or to specify a custom row–supply procedure to use as an alternative to JAM/DB*i*.

After you have modified `rwmain.c` and/or `rwopts.c`, run the supplied makefile[3] to create the new report generator executables. Refer to the makefile for instructions on installing user-written functions and selecting the executable modules to be made.

Refer to Section 4.5 for further information on installing your own functions into the ReportWriter executable.

B.2
FETCHING INTO ONSCREEN ARRAYS

When fetching data into onscreen arrays, you must be aware of the way your input function handles the last fetch, which may or may not contain enough rows to fill the array.

This section explains what happens on the final fetch when ReportWriter uses JAM/DB*i* to retrieve rows from a database. If you are using a developer-written input function in place of JAM/DB*i*, you may want to review this section anyway to help identify issues that need to be considered when the target variables are onscreen arrays.

As ReportWriter cycles through the detail query, JAM/DB*i* returns ***n*** rows from the database, where ***n*** is the number of elements in the smallest target variable array. There is no assurance, however, that the number of rows satisfying the query will be a multiple of the number of elements in the target arrays. Thus, the last cycle may yield fewer than ***n*** rows.

Beginning with JAM/DB*i* release 5.1, JAM/DB*i* provides full information to ReportWriter on completely or partially filled dsetination arrays. When linked with JAM/DB*i* release 5.1 or later, ReportWriter handles all queries without loss of data, whether or not the destination variables are arrays.

When linked with a release of JAM/DB*i* previous to 5.1, however, ReportWriter ignores the last fetch if the destination arrays are not completely filled. Suppose, for example, the target variables for a fetch are arrays of three elements each. If seven rows of the database satisfy the query, the first two cycles would fully populate the arrays, and the third would yield a single row (which would normally be lost):

```
fetch1:     row1.....
            row2.....
            row3.....
```

3. The name of the makefile supplied with ReportWriter depends upon your operating system. It generally contains the word "make" in the file name The function of this file is to create (make) the ReportWriter executables. Since the procedure is operating system dependent, each makefile contains the applicable instructions for its operation.

```
fetch2:     row4.....
            row5.....
            row6.....

fetch3:     row7.....
```

The following sections describe two ways to ensure that the results of the last fetch are retained: moving the output to a break footer or padding the source table with trailing blank rows.

Of the two proposed solutions, the first, using a break footer, is the more generally applicable and is, therefore, the recommended technique.

NOTE: The methods described in the following sections should be used only if ReportWriter is linked with JAM/DB*i* release 5.0 or earlier.

B.2.1

Outputting the Array in a Break Footer

Create a report area containing the array you want to use for output of the query results. Instead of using this as the detail output area, however, you will use it as a break footer. In the `detail` statement, fetch one row at a time (into single–element fields), but do not perform output. The detail JPL should copy the results of the fetch into the next available elements of the arrays in the break footer area.

Your detail processing will need to keep count of the number of rows fetched since the break footer was output. The count variable will serve two purposes: it will tell ReportWriter which elements to copy the current row into, and it can be used to update the break field: break every *n* rows (where *n* is the number of elements in the onscreen arrays) to output the arrays.

No rows are lost, since any partially filled arrays are output as the final break footer.

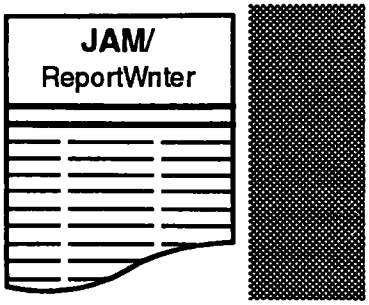
B.2.2

Padding the Source Table

This technique is not suitable for all databases or all applications. It is useful in some situations and is, therefore, noted here.

Pad the source table with *n*–1 trailing blank rows, where *n* is the size of the query's destination arrays. The added rows will blank–fill the final fetch and ensure that the final row containing actual data is passed to ReportWriter.

There may be some difficulty with this approach: you may be unable to modify the tables, the query may perform (inner) joins, a stored procedure may be used to invoke the fetch, etc. Using temporary tables may resolve some of these problems.



Appendix C
Troubleshooting Guide

When ReportWriter detects an error during report compilation or at runtime, it sends an appropriate message to the standard error file on your system. ReportWriter's error messages will help you detect and correct these errors involving the mechanics of using ReportWriter.

You may also encounter errors that show up as unexpected results in the report format, content, or output. The chart in this appendix is intended to help you identify and correct problems in report composition and output specification.

Problem	*Cause*	*Solution*
A field fetched by the `detail` query is blank or data is in the wrong format.	Field name differs from name of fetched column	Correct the spelling of the field name.
		Use `SELECT` aliasing; refer to the JAM/DB*i* manual for instructions.
	JAM/DB*i* release 4 is returning dates in the DBMS's native format.	Create, either on the report format screen or in the LDB, additional variables whose date formats match that of your DBMS. Fetch into those variables and use a JPL procedure invoked in the `detail` statement to copy them into the appropriate fields in the desired report areas. (For information on variable and date formats, refer to descriptions of the `math` statement and the `@date` function in the *JPL Guide* of the JAM documentation set.)
	The field has the `no display` attribute.	Turn off the `no display` attribute for this field via JAM's Display Attributes window.
	The field appears on a line that does not bear the correct detail area name tag.	Add the correct name tag to the line containing this field, or move the field to a line in the desired report area.

continued...

Problem	*Cause*	*Solution*
The report stops processing a detail query after one row.	Another database fetch, using the default `SELECT` cursor, is being run—in a JPL or C routine invoked from the report script, or in a screen entry/exit procedure.	Make sure that all other SQL statements executed during report generation use their own named cursors.
Number of rows reported from the database is less than the number expected.	ReportWriter is linked with **JAM/DB*i*** release 5.0 or earlier, and destination variables are onscreen arrays; last fetch does not fill the array and is not passed to ReportWriter.	Fetch into single-element fields.
		Use one of the workarounds suggested in Section B.2.
An array will not shrink, even though the `area` clause was issued with the `shrink` keyword.	There is another field or display text on the same line as the unwanted array element.	Remove all other fields and display text from lines you want to be affected by `shrink`.
	An element further down in the array contains data. ReportWriter shrinks out only trailing array elements.	Consolidate populated array elements so that the blank elements are at the end.

continued...

Problem	*Cause*	*Solution*
Break header or footer data is out of sync with its break group.	Procedures that calculate break footer values are improperly placed in the `detail` statement.	Procedures that calculate running totals or other break footer values should be invoked after break processing in the `detail` statement. (Refer to Section 7.2.3 for a further explanation of this topic.)
	Procedures that calculate break header or footer values are improperly placed in the `break` statement.	Break header or footer procedures that fetch or calculate data to be output in the respective header or footer must be invoked before the corresponding `area` clause.
	A non–break field in the break footer is showing its post–break, rather than pre–break value.	Define the field as a lowest–level break field with no output or actions. (Refer to Section 5.2.3 for information on retaining pre–break values and to Section 5.1.2 for an explanation of how to use the `breakcheck` keyword to ensure proper break footer output.)
Break header or footer data that was correct in a row–level report is out of sync when the report is converted to a summary report.	Removal of the `area` clause from the `detail` statement can change the point at which break checking and processing occur.	Insert the `breakcheck` keyword at the point where break checking and processing should take place.

continued...

Problem	*Cause*	*Solution*
When you attempt to run your reports after upgrading them to release 5.1, ReportWriter gives the message "Error reading ***reportfile.***"	The version of `rprt2bin` that compiled the report is later than the version of ReportWriter you are running.	Make sure that the executable you are using (`rwrun`, `jamrw`, `jxrw`, or your own ReportWriter application) was made from files of the same ReportWriter release as the `rprt2bin` you used to compile the report format screens.
JAM/DB*i* errors mention `dbms declare` statements that do not appear in the application's JPL code.	ReportWriter implements the `query` clause of the `detail` statement as a `dbms declare` statement. It creates for each report a cursor whose name is `_RWcursor4`***n***, where ***n*** is the report name (if the report is named). Each error message (such as "SQL parse error") that refers to one of these cursor names reflects an error in a query of the associated report.	Correct the query.
		continued...

Problem	*Cause*	*Solution*
An area name tag referenced in the script is not found when the report format screen is compiled.	The area name has a JAM draw field symbol embedded in it. For example: `emp_lst`.	Remove underbars (the JAM default draw field symbol) from area names. If you have any other draw field symbols defined for the report format screen, do not use them, either, in area names. (Wherever compound area names appear in this manual, hyphens are used in order to avoid underbars. For example: `emp-lst`.)
	The referenced area is defined on another JAM screen.	Use the `include screen` compiler directive to incorporate all required screens into the report.
	The area name is misspelled, either in the script or in the name tag; or the area name tag does not appear on the screen.	Make sure that both the spelling and case of the area name are identical in the script and on the screen layout.
		Make sure that the referenced area has been defined and that its name tag has been appended to each line of the area.

continued...

Problem	*Cause*	*Solution*
An area name tag referenced in the script is not found when the report format screen is compiled. *(continued)*	The report format screen was created by `rw4to5`, and the original script uses the file name extension `.jam` in the specification of form (area) names. By default, `rw4to5` drops this extension when it creates area name tags, so that scripts that use only the base file names will work properly.	In `jxform`, edit the script in the report format screen to remove each use of `.jam` where it appears in an area name specification.
		Rerun `rw4to5` with the `-k` option to retain the extension in the generated area name tags.
Parameters work fine in a `report` clause but produce error messages when used in a `cursor` clause.	The two clauses employ different syntax for parameter specification. In a `report` clause (as well as in a `jpl` or `call` clause), parameters are separated from the object called and from one another by one or more spaces. In a `cursor` clause, a space follows the name of the cursor; the parameters that follow must be listed according to the syntax of the `dbms execute` statement of the **JAM/DB*i*** in use. This is usually a list of variable names (without preceding colons) separated by commas.	Use space-separated strings or colon-expanded variables for `report` parameters. Use comma-separated simple variable names for `cursor` parameters.

continued...

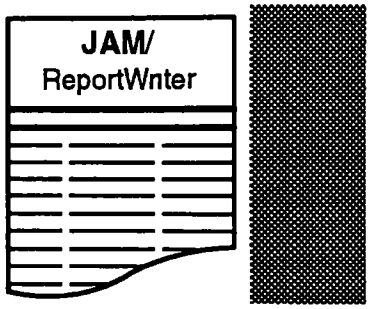
Problem	*Cause*	*Solution*
A C (or other supported language) function cannot be called from the report script.	The function either has not been entered into `funclist.c` or has not been installed. (Refer to Section 4.5 for additional information on installing called functions.)	Make sure the function is on the control function or prototyped function list of `funclist.c`.
		Make sure the call to `sm_do_uinstalls` has been uncommented in `rwmain.c` (if using `rwrun` stand-alone).
		Make sure the executable has been rebuilt since source changes were made.
Report output cannot be sent to the printer.	No device file has been specified, or the device file in use does not contain a `spool` parameter to direct the output to the printer.	1. Create a device file containing the appropriate `spool` parameter. 2. Compile the device file with the `dev2bin` utility. 3. Reference the device file at runtime (`-d` ***devfile*** option on the command line or in `RWOPTIONS`). (Refer to Section 8.1 for more information on device configuration files and to Section 9.4 for information on `RWOPTIONS`.)
	The spool command specified is incorrect.	Verity that the spool command specified in the device file is correct for your environment.

continued...

Problem	*Cause*	*Solution*
The printer does not reset to the beginning of the next page when done printing a report.	ReportWriter does not issue a form feed at the end of output.	Use the `reset` parameter in the device configuration file to force a form feed at the end of the report: `reset = \014` (Refer to Section 8.1 for more information on device configuration files.)
The number of lines and/or columns is different than expected.	Default settings are in effect. ReportWriter's default page size is 60 lines by 132 columns.	Specify the required `lines` and `columns` settings in either an `init` statement or a device configuration file.
	Conflicting page sizes are specified in the device configuration file and in an `init` statement. If both are specified, the values in the `init` statement take precedence over the values in the device file.	If you want to be able to change the page size specifications for different runs of the report, remove them from the `init` statement and use, instead, device files with the required page size parameters. (Refer to Section 8.1 for more information on device configuration file parameters.)
		If you want the page size specification to be consistent for all runs of the report, put the `lines` and `columns` parameters in an `init` statement in the report script. (Refer to Section 5.4 for more information on output parameters in the `init` statement.)

continued...

Problem	*Cause*	*Solution*
Core dump occurs while accessing the database through JAM/DB*i* release 5.	Some beta versions of JAM/DB*i* release 5 will core dump if the database engine is not properly initialized.	Always use your JAM/DB*i* makefile to create the ReportWriter executables, where possible. Otherwise, modify the ReportWriter makefile to link all the modules required, particularly `dbiinit.o`, if it exists, to make JAM/DB*i*. `dbiinit.o` initializes the database engine. If your JAM/DB*i* release 5 distribution does not include `dbiinit.o`, modify your `rwmain.c` source module to manually initialize the database engine—see the instructions in `rwmain.c` and use the `jmain.c` in your JAM/DB*i* distribution as a model.
A report format screen is displayed unexpectedly in a `jamrw` application.	JAM flushes all active screens to the display while posting messages; this includes screens not intended for display (such as report format screens).	Don't use the JPL statement `msg` or the screen manager function `sm_emsg ()` or its variants from within programs that are invoked by `jpl` or `call` clauses in the report script. Instead, save the message in a non–output field or in an LDB variable, and display it after execution of the `rwrun` statement.



Appendix D
Examples

The sample reports in this appendix demonstsrate some of the techniques explained in the body of this manual. Each example begins with a prose description of the report and a list of the techniques illustrated. Sample output is shown, followed by a listing of the screen JPL module, the screen layout, and the field descriptions.

The JAM `f2asc` and `lstform` utilities were used to capture the JPL, layout, and field listing of each example. The output of these utilities is reproduced here with minimal embellishment.

Procedures for connecting to the database are not shown in these examples, both because the report may be part of a JAM/DB*i* application that manages the connection and also because the syntax for accomplishing this in JAM/DB*i* depends upon the version of JAM/DB*i* and the database engine in use.

D.1
SAMPLE APPLICATION – REVISITED

Chapter 3 of this manual presents a simple report based on the sample application described in the JAM and JAM/DB*i* documentation. This section shows how you might enhance this relatively basic report.

The following features are added to the report originally presented in Section 3.3.1:

- Page footers on all pages except the title page
- Page numbers (field `page` and JPL procedure `incr_page`); example of a procedure that must be executed before the corresponding area is output

- Trailer page with a grand total of all salaries reported
- Break footer omitted if the break group has only one entry (`noorphanbreak` keyword)
- Break headers
- Break field (`grade`) displayed only in the first detail area of the break group (`norepeat` keyword)
- Non–output fields (`last` and `first`) on the report format screen: condensing the employee's first and last names into a single field (field `name` and JPL procedure `condense_name`); another example of a procedure that must be executed before the corresponding area is output

Sample output from this report is shown in Figure 18.

```
                    XYZ Corporation

                  Personnel Department

              Report of Employee Salaries by Grade
```

```
GRADE       SSN              NAME                    SALARY
-----    -----------  ------------------------   ----------
  B      122-99-4102  Jones, Michael              26,000.00

GRADE       SSN              NAME                    SALARY
-----    -----------  ------------------------   ----------
  C      038-68-6826  Jones, Barnaby             29,500.00

GRADE       SSN              NAME                    SALARY
-----    -----------  ------------------------   ----------
  D      139-42-1651  Blake, Norman              89,500.00
         154-32-6610  Cory, Richard              43,100.00
         310-77-3997  Grundy, Janet              38,000.00
         310-32-0084  Jones, John P.             47,500.00

                    Total salaries at GRADE D   218,100.00

GRADE       SSN              NAME                    SALARY
-----    -----------  ------------------------   ----------
  E      122-98-6541  Aumond, Hilary             37,800.00


         Report of Employee Salaries by Grade         page 1
```

```
     ************************************************
     *                                              *
     *     Total Salaries For All Grades Reported   *
     *                                              *
     *                   $311,400.00                *
     *                                              *
     ************************************************


       Report of Employee Salaries by Grade         page 2
```

Figure 18: Output of Salary Report

The screen JPL module is reproduced below. Additions to the report originally presented in Chapter 3 are printed in boldface type.

The first part of the JPL module contains the report script:

```
# << begin report >>
#
# init jpl     = startup  /* invoke the JPL proc "startup" */
#      area    = titlepg  /* output the report title page */
#      newpage            /* ensure that next output after title page
#                            begins on a new page */
#
# /* page footer should not appear on the title page; hence, the
#    following page statement is placed after the title page has been
#    output; page footers will appear on all subsequent pages: */
#
# page footer jpl  = incr_page   /* page number must be incremented*/
#             area = pfoot       /* before the pg footer is output */
#
# break field  = grade    /* group the report output by employment
#                            grade */
#      header  area   = bhead
#      footer area    = bfoot
#             jpl     = reset_tot  /* after break footer area is
#                                     output, invoke JPL procedure to
#                                     reset the cumulative salary
#                                     total */
#      norepeat           /* display grade only for the
#                            first row of each break group */
#      noorphanbreak      /* do not output break footer if there is
#                            only one row in the break group */
#
# detail query = "SELECT emp.ssn, emp.last, \
#          emp.first, emp.grade, acc.sal \
#          FROM emp, acc \
#          WHERE emp.ssn = acc.ssn \
#          ORDER BY emp.grade, emp.last, \
#                emp.first"
#        jpl  = condense_name /* must be invoked before detail
#                                area is output */
#        area  = employee
#        jpl   = add_salary      /* invoke JPL procedure to maintain
#                                   running total of salaries for
#                                   the current break group (grade)
#                                   and for all employees reported */
#
# insert newpage
#
# insert   area    = trailer
# /*       jpl     = cleanup  (omitted in this example) */
#
# << end report >>
```

The JPL procedures referenced in the report script follow in the JPL module:

```
proc startup
    # This procedure is invoked once, at the start of report
    # generation. It performs two functions:
    #   1) initializes variables used in the report
    #   2) opens a connection to the database (code for this
    #      function is omitted in this example)

    # the following statement initializes the running
    # total of salaries for the break group (grade)
  cat sal_tot

    # the following statement initializes the running
    # total of salaries for all employees reported
  cat all_sal

    # page number: will be incremented each time the page footer is
    # output
  cat page


proc reset_tot
    # After the break footer has been output, the
    # cumulative salary total must be reset for the
    # next break group

  cat sal_tot


proc add_salary
    # Add the salary for this employee to the
    # running total of salaries at this grade
    # and to the running total of all salaries reported

  math sal_tot = sal_tot + sal
  math all_sal = all_sal + sal


proc incr_page
    # compute the page number immediately before the page
    # footer is output

   math %.0 page = page + 1

proc condense_name
    # to close up the gap between the last and first names,
    # consolidate them into a single field

   cat name last ", " first
```

The screen layout for this report example is shown in Figure 19.

```
                    XYZ Corporation                          << titlepg
                                                             << titlepg
                  Personnel Department                       << titlepg
                                                             << titlepg
            Report of Employee Salaries by Grade             << titlepg

                                                             <<bhead
GRADE       SSN              NAME                SALARY      <<bhead
-----    -----------  -------------------------  ----------  <<bhead

  _      __________   _________________________  _________   <<employee

                                                             <<bfoot
                 Total salaries at GRADE _      _________   <<bfoot
                                                             <<bfoot
                                                             <<bfoot

          *******************************************        <<trailer
          *                                         *        <<trailer
          *   Total Salaries For All Grades Reported*        <<trailer
          *                                         *        <<trailer
          *                 __________              *        <<trailer
          *                                         *        <<trailer
          *******************************************        <<trailer

          Report of Employee Salaries by Grade        page __  <<pfoot

    The following two fields are non-output fields  Their field names are
    "last" and "first", respectively:
         ____________________          ___________
```

Figure 19: Salary Report Screen Layout

Field descriptions, as generated by JAM's `f2asc` utility, are reproduced below:

```
F:grade
#  NUMBER=1
   LINE=11 COLUMN=6
   UNFILTERED
   LENGTH=1 ARRAY-SIZE=1
   WHITE HILIGHT

F:ssn
#  NUMBER=2
   LINE=11 COLUMN=13
   UNFILTERED
   LENGTH=11 ARRAY-SIZE=1
   WHITE HILIGHT
```

```
F:name
#  NUMBER=3
   LINE=11 COLUMN=26
   UNFILTERED
   LENGTH=25 ARRAY-SIZE=1
   WHITE HILIGHT

F:sal
#  NUMBER=4
   LINE=11 COLUMN=54
   UNFILTERED
   LENGTH=10 ARRAY-SIZE=1
   WHITE HILIGHT
   RIGHT-JUSTIFIED
   CURR-FORMAT= DEC-SYMBOL= . MIN-DEC-PLACES=2 MAX-DEC-PLACES=2
     ROUND-ADJUST RIGHT-JUST

F:grade.1
#  NUMBER=5
   LINE=14 COLUMN=48
   UNFILTERED
   LENGTH=1 ARRAY-SIZE=1
   WHITE HILIGHT

F:sal_tot
#  NUMBER=6
   LINE=15 COLUMN=54
   UNFILTERED
   LENGTH=10 ARRAY-SIZE=1
   WHITE HILIGHT
   RIGHT-JUSTIFIED
   CURR-FORMAT= DEC-SYMBOL= . MIN-DEC-PLACES=2 MAX-DEC-PLACES=2
    ROUND-ADJUST THOU-SEP-SYMBOL= , RIGHT-JUST

F:all_sal
#  NUMBER=7
   LINE=23 COLUMN=30
   UNFILTERED
   LENGTH=12 ARRAY-SIZE=1
   WHITE HILIGHT
   CURR-FORMAT= DEC-SYMBOL= . MIN-DEC-PLACES=2 MAX-DEC-PLACES=2
    ROUND-ADJUST THOU-SEP-SYMBOL= , CURR-SYMBOL= $ CUR-LEFT RIGHT-JUST

F:page
#  NUMBER=8
   LINE=27 COLUMN=63
   DIGITS-ONLY
   LENGTH=2 ARRAY-SIZE=1
   WHITE HILIGHT
```

```
F:last
#  NUMBER=9
   LINE=28 COLUMN=12
   UNFILTERED
   LENGTH=20 ARRAY-SIZE=1
   WHITE HILIGHT

F:first
#  NUMBER=10
   LINE=28 COLUMN=43
   UNFILTERED
   LENGTH=12 ARRAY-SIZE=1
   WHITE HILIGHT
```

D.2
USING THE `tbl2r` UTILITY

ReportWnter provides a utility that creates, from a specified database table, a report format screen that can be used to generate a simple report showing the contents of the table. The report format screen produced by this utility can also be used as the starting point for a more elaborate report.

The examples in this section develop three reports:

- The first is generated, without modification, from the report format screen produced by the `tbl2r` utility.
- The second is a cosmetic enhancement only to the original report layout. No changes were made to the report script.
- The third report is the result of modifications to both the screen layout and the report script. Several JPL procedures were also added to effect the required processing.

The reports shown in this section are based on the `emp` database table, described in Chapter 3, "Sample Application."

D.2.1
A Quick Start Report

This section describes the report format screen, `emp.jam`, produced when the `tbl2r` utility is run on database table `emp`. The report generated from this report format screen is shown in Figure 20. It is a simple listing of the contents of the database table. The page

header lists the table column titles.

```
ssn         last                first        street
city            st zip   grade

038-68-6826 Jones               Barnaby      321 West 11 St
Albuquerque     NM 9876  C
122-98-6541 Aumond              Hilary       11-12 Front St
Albuquerque     NM 09876 E
122-99-4102 Jones               Michael      5 Maple Drive
Albuquerque     NM 09876 B
139-42-1651 Blake               Norman       34 Concord Ave
Albuquerque     NM 09876 D
154-32-6610 Cory                Richard      411 Ann St
Albuquerque     NM 09876 D
310-77-3997 Grundy              Janet        70-2 Poe St
Albuquerque     NM 09876 D
310-32-0084 Jones               John P.      9 Vern Terrace
Albuquerque     NM 09876 D
```

Figure 20: Output of Report `emp.jam`

The screen JPL module, reproduced below, consists only of a report script. There are no JPL procedures.

```
# <<BEGIN REPORT>>
# page header area = header
#
# detail area = detail
#       query = "select * from emp"
# <<END REPORT>>
```

Figure 21 shows the screen layout for this report.

Field descriptions, as generated by JAM's `f2asc` utility, are reproduced below:

```
F:ssn
#  NUMBER=1
   LINE=5 COLUMN=2
   UNFILTERED
   LENGTH=11 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
```

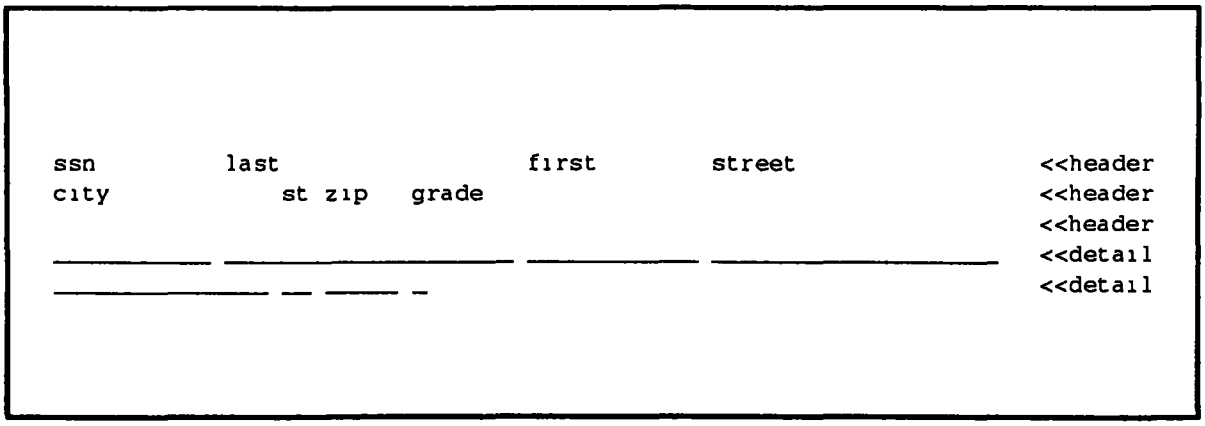


Figure 21: Report Format Screen `emp.jam` – Output Areas

```
F:last
#  NUMBER=2
   LINE=5 COLUMN=14
   UNFILTERED
   LENGTH=20 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:first
#  NUMBER=3
   LINE=5 COLUMN=35
   UNFILTERED
   LENGTH=12 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:street
#  NUMBER=4
   LINE=5 COLUMN=48
   UNFILTERED
   LENGTH=20 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:city
#  NUMBER=5
   LINE=6 COLUMN=2
   UNFILTERED
   LENGTH=15 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:st
#  NUMBER=6
   LINE=6 COLUMN=18
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
```

```
F:zip
#  NUMBER=7
   LINE=6 COLUMN=21
   UNFILTERED
   LENGTH=5 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:grade
#  NUMBER=8
   LINE=6 COLUMN=27
   UNFILTERED
   LENGTH=1 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
```

D.2.2

A Cosmetic Improvement

The report shown in this section is built on the report format screen produced by `tbl2r` and described in the previous section. To create the output shown in Figure 22, the developer

- rearranged the fields in the `detail` area and
- changed the display text in the `header` area.

No other changes were made to the report format screen. The script was not modified in any way, and no JPL procedures were added.

The new layout of the report format screen is shown in Figure 23.

```
SSN          EMPLOYEE                                  GRADE
-----------  ----------------------------------------  -----

038-68-6826  Jones                , Barnaby             C
             321 West 11 St
             Albuquerque     NM 9876

122-98-6541  Aumond               , Hilary              E
             11-12 Front St
             Albuquerque     NM 09876

122-99-4102  Jones                , Michael             B
             5 Maple Drive
             Albuquerque     NM 09876

139-42-1651  Blake                , Norman              D
             34 Concord Ave
             Albuquerque     NM 09876

154-32-6610  Cory                 , Richard             D
             411 Ann St
             Albuquerque     NM 09876

310-77-3997  Grundy               , Janet               D
             70-2 Poe St
             Albuquerque     NM 09876

310-32-0084  Jones                , John P.             D
             9 Vern Terrace
             Albuquerque     NM 09876
```

Figure 22: Output of Report `emp2.jam`

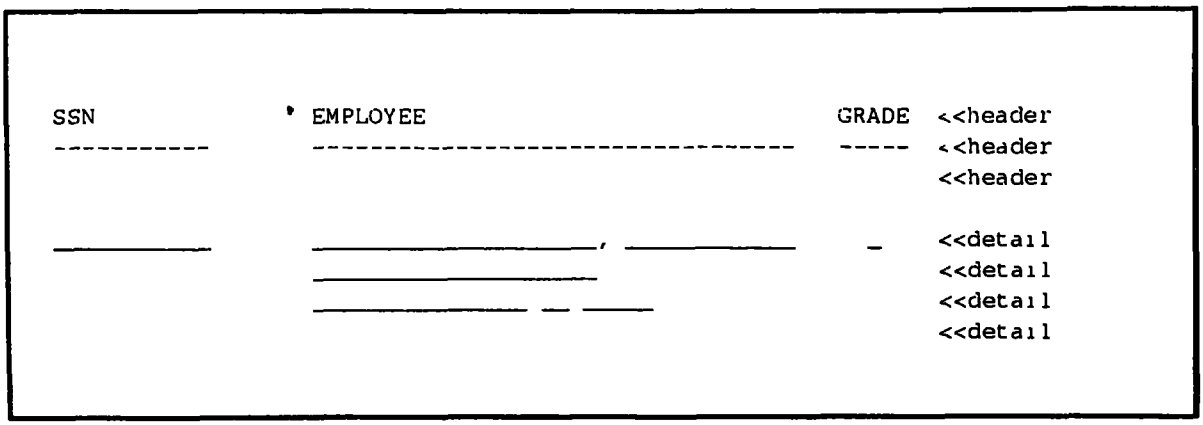


Figure 23: Report Format Screen `emp2.jam` – Output Areas

D.2.3

More Extensive Changes

The report shown in this section is also built on the report originally generated by the `tbl2r` utility. `tbl2r` provided a headstart on the report development process by creating a report format screen that included fields of the appropriate types and lengths for the columns in the database table and by writing a rudimentary script with a simple `query` clause. The developer can now retain or discard fields as necessary, add new fields, define breaks, modify the query, and so forth, to create the desired report.

The report shown in Figure 24 lists employees alphabetically. It shows the last name, first name, social security number, and employment grade level for each employee listed in the table. A blank line is inserted after every third row to improve readability of the report.

This revision adds the following features to the original report

- Computed break (field `rowgroup`; to insert the blank line as a break footer after every third line of the detail area)
- Non–output fields on the report format screen:

  `last` and `first` to receive the employee's first and last names, which are then condensed into a single field (field `emp_name` in the `detail` area and JPL procedure `condense_name`)

  `rowcount` and `rowgroup` to effect the computed break

```
EMPLOYEE                          SSN           GRADE
--------------------------------  -----------   -----

Aumond, Hilary                    122-98-6541     E
Blake, Norman                     139-42-1651     D
Cory, Richard                     154-32-6610     D

Grundy, Janet                     310-77-3997     D
Jones, Barnaby                    038-68-6826     C
Jones, John P.                    310-32-0084     D

Jones, Michael                    122-99-4102     B
```

Figure 24: Output of Report `emp3.jam`

- Specifying the order in which rows are to be fetched (`order by` clause added to the query)
- Procedure that must be invoked before area output (`condense_name`)
- Procedure invoked after break checking (`do_rows` to compute the row number and change the break field value as needed)

In addition, unneeded fields from the original screen were deleted, and a new output area was added.

The new area, `blank-line`, is specified as the break footer and is the blank line output after every third row. Note that this area name contains a hyphen rather than an underbar between the two words of the name. By default, the JAM screen editor interprets underbars in area name tags as screen fields; area names with underbars (or any other draw field symbol in use) are not correctly recognized when the screen is compiled.

The screen JPL module is reproduced below. Additions to the report originally created by `tbl2r` are printed in boldface type.

```
# <<BEGIN REPORT>>
#
# insert jpl = startup
# page header area = header
#
# /* The break field shown below is a computed break.
#    Its value is computed in the JPL proc do_rows, which
#    is invoked during detail processing. */
#
#
```

```
# break field = rowgroup
#        footer area = blank-line
#               jpl = reset_row_count
#
# detail  jpl = condense_name
#        area = detail
#         jpl = do_rows
#       query = "select * from emp order by last, first"
#
# <<END REPORT>>

proc startup
  #  This procedure initializes the variables required for
  #  computing the break field.

  cat rowcount
  cat rowgroup

proc condense_name
  #  To close up the gap between last and first name, consolidate
  #  them into a single field

  cat emp_name last ", " first

proc do_rows
  #  Compute the position of the just-output row within its break
  #  group. If it is the third row, then also increment the value of
  #  the break field, rowgroup, so that a break will occur on the next
  #  fetch.

  math rowcount = rowcount + 1
  if rowcount == 3
     {
        math rowgroup = rowgroup + 1
     }

proc reset_row_count
  #  After the break footer has been output, reset the row count
  #  so that the do_rows procedure will work properly for the next
  #  set of rows output

  cat rowcount
```

Figure 25 shows the screen layout for this report.

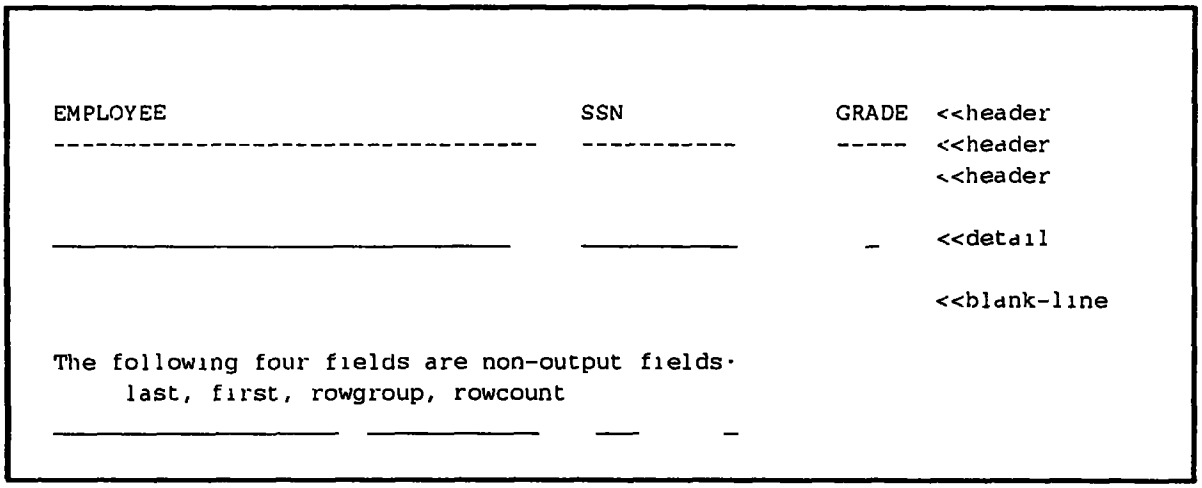


Figure 25: Report Format Screen `emp3.jam` – Output Areas

Field descriptions, as generated by JAM's `f2asc` utility, are reproduced below:

```
F:emp_name
#   NUMBER=1
    LINE=6 COLUMN=2
    UNFILTERED
    LENGTH=32 ARRAY-SIZE=1
    WHITE HILIGHT

F:ssn
#   NUMBER=2
    LINE=6 COLUMN=39
    UNFILTERED
    LENGTH=11 ARRAY-SIZE=1
    WHITE UNDERLINE HILIGHT

F:grade
#   NUMBER=3
    LINE=6 COLUMN=59
    UNFILTERED
    LENGTH=1 ARRAY-SIZE=1
    WHITE UNDERLINE HILIGHT

F:last
#   NUMBER=4
    LINE=12 COLUMN=2
    UNFILTERED
    LENGTH=20 ARRAY-SIZE=1
    WHITE UNDERLINE HILIGHT
```

```
F:first
#  NUMBER=5
   LINE=12 COLUMN=24
   UNFILTERED
   LENGTH=12 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:rowgroup
#  NUMBER=6
   LINE=12 COLUMN=40
   UNFILTERED
   LENGTH=3 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:rowcount
#  NUMBER=7
   LINE=12 COLUMN=49
   UNFILTERED
   LENGTH=1 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
```

D.3
SUBREPORTS

This section contains two subreport examples.

The first builds a subreport in stages, beginning with a report that can be invoked on its own as a primary report and then adding enhancements that invoke the original report as a subreport.

The second example shows how the simulated subreport application in Chapter 10 would be implemented as a true subreport if JAM/ReportWriter release 5.1 were linked with JAM release 5.03a or higher and JAM/DB*i* release 5.

D.3.1
Comprehensive Subreport Example

The example in this section builds an application for selecting and printing the resumes of opera singers by ID, by voice range, and for all singers in the database.

The application is built in stages, beginning with a single report script containing a primary report and a subreport; this report generates the resume for one singer and serves as the base for additional reporting capabilities.

The initial report and its enhancements provide examples of

- nested subreports,
- reports that can be invoked as either primary reports or subreports,
- subreports stored in screens external to the primary report format screen, and
- a variety of methods for passing arguments to both primary and subreports.

Stage 1: Print the resume of an opera singer identified by ID number. The report name is `resume`; it takes a single argument, the ID of the singer whose resume is to be generated. The following command line invocation sends the output to a file named `resume.txt`; the resume generated will be for the singer whose ID is `47`.

```
rwrun resume -o resume.txt 47
```

The report uses three separate queries to format fetched data for the resume and a subreport to print the singer's name and address at the top of each page.

The destination variable of the first query, `intro`, is a word-wrapped array. The `shrink` keyword, as usual, removes its unused elements.

Only the third query uses break processing. The `clear breakspecs` statement isolates the break statements from the previous queries.

The script for this report is:

```
# << begin report = resume >>
#
# init parameter = id
#      lines = 40
#      jpl = logon
#
# insert jpl = clearcontinued
#
# page header report = "address :id"
#      footer jpl = setcontinued
#
# detail query = 'select intro from allintros where id = :+id'
#        area = intro shrink
#
# insert area = e-begin
#
# detail query = 'select estart, eend, school from edu \
#                 where student = :+id order by estart'
#        area = e-detail
#
# insert area = r-begin
#
# clear breakspecs
#
```

```
# break field = company
#       header area = r-where showattop
#
# break field = opera
#       norepeat
#       footer area = r-foot
#
# detail query = 'select company, opera, role from performances \
#                  where singer = :+id order by company, opera'
#        area = r-detail
#
# << end report >>
#
# << begin report = address >>
#
# init parameter = id
#
# break field = name
#       header area = h-begin
#       footer area = h-end
#
# detail query = 'select name, phone, addrseq, addr from nameaddr\
#                  where id = :+id order by addrseq'
#        area = h-detail
#
# << end report >>
```

The JPL procedures invoked by this script follow:

```
proc logon
if loggedon != 'y'
{
#   dbms-specific logon statment goes here
      cat loggedon 'y'
}

proc clearcontinued
cat continued

proc setcontinued
cat continued "(continued)"
```

The screen layout for the `resume` report is shown in Figure 26.

Stage 2: We can also invoke the preceding report as a subreport by including it in another report format screen. This example invokes the resume report for each singer having a range given by the primary report's argument. The report name is `onerange`.

A header page prints the chosen voice category. The report format screen field `voice` also serves as a parameter field. The `preserve initspecs` keyword in the subreport invocation ensures that it takes on the page size of the parent report.

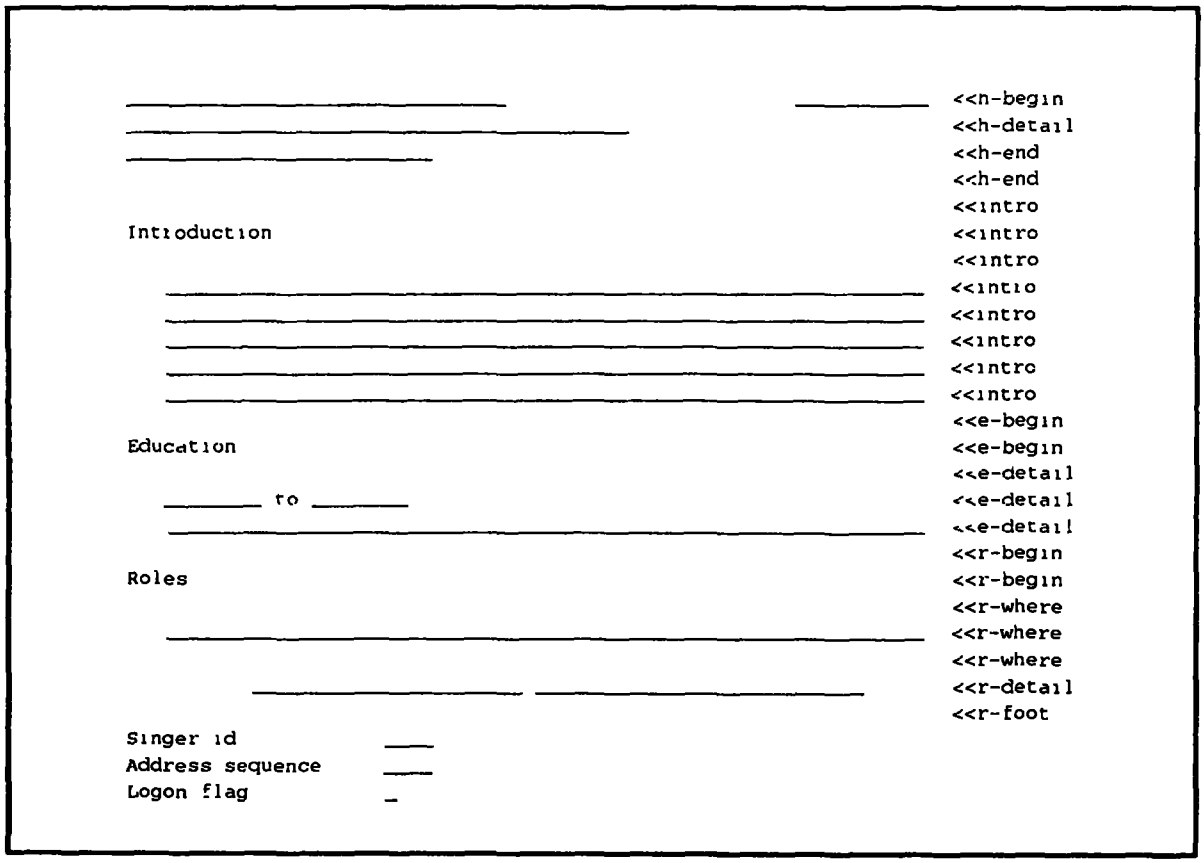


Figure 26: Report `resume.jam` – Screen Layout

We could invoke the report `onerange` from the command line as

```
rwrun onerange -o resume.txt Soprano
```

or from JPL, as in

```
cat RWOPTIONS 'Soprano'
rwrun onerange resume.txt
```

The script for this report is

```
# << begin report = onerange >>
# << include screen = resume >>
#
# init lines = 30
#      parameter = voice
#      jpl = logon
#
# insert area = title
#        newpage
#
# detail query = 'select id from allintros \
#                 where range = :+voice'
```

```
#         report = "resume :+id"
#                  preserve initspecs
#         newpage
#
# << end report >>
```

The screen layout for the `onerange` report is shown in Figure 27.

```
                                          <<title
                  Hubris Management Company   <<title
                                          <<title
                  Resumes  _____________   <<title
```

Figure 27: Report `onerange.jam` – Screen Layout

Stage 3: The report `onerange`, as well, can serve as a subreport. Our final example includes it and fetches all voice ranges to produce resumes for all clients.

The field `range` on the report format screen is used as the query destination and also for passing an argument to the subreport. All output is performed in the subreport; no `preserve` keywords are needed in this example.

```
# << begin report >>
# << include screen = onerange >>
#
# init jpl = logon
#
# detail query = 'select distinct range from allintros \
#                 order by range'
#         report = "onerange :range"
#
# << end report >>
```

Sample output from this application is shown in Figure 28 through Figure 30.

```
                    Hubris Management Company

                    Resumes   Soprano
```

```
Alma Gluck
14 Willow Lane
Scarsdale, New York
555-4321

Introduction

   A promising young soprano

Education

   01/01/81 to 01/01/91
   Actors' and Singers' School

Roles

   Acme Industrials

         The World of Fiber     Celery

   Stony Creek Community Theater

         Carousel               Julie
```

Figure 28: Resume Application – Sample Output

```
Maria Stupendita
23 Rue Pons
Paris
046 554 22-43

Introduction

   A versatile soprano with stage presence   Skilled interpreter
   of Verdi heroines, and a proponent of bel canto

Education

   09/01/71 to 06/01/73
   Paris Conservatory

   07/01/73 to 11/15/75
   Juilliard School

Roles:

   Berlin Opera

          La boheme                Mimi

          Les contes d'Hoffmann    Antonia

   Covent Garden

          Aida                     Aida
```

Figure 29: Resume Application – Sample Output (cont'd.)

```
Maria Stupendita                                        (continued)
23 Rue Pons
Paris
046 554 22-43

   Covent Garden

         Don Carlo              Elisabetta

         La Boneme              Mimi
                                Musetta

         Otello                 Desdemona

         Tannhauser             Elizabeth
                                Venus

         Wozzeck                Marie

   Rome Opera

         Fidelio                Leonora

         Norma                  Adalgisa
                                Norma
```

Figure 30: Resume Application – Sample Output (cont'd.)

D.3.2

Example from Chapter 10—Revisited

Section 10.1 of this manual shows a sample report that uses super–query and sub–query techniques to produce subreports. These are the techniques that were used to produce subreports under ReportWriter release 5.0. Users of the current release of ReportWriter may still have to use these techniques if it is not linked with JAM release 5.03a or higher and JAM/DB*i* release 5.

The example in this section shows how the subreport application shown in Section 10.1 would be implemented with the full subreport capability provided in ReportWriter 5.1.

Developers who have previously used ReportWriter 5.0 will notice that this example also takes advantage of other features new to release 5.1, namely:

- In procedure `get_sum_bal`, a `sql` statement can be used because ReportWriter, when linked with JAM/DB*i* release 5 or later, now uses named cursors for its queries.
- Multiple `area` and/or `report` clauses appear in a single statement in both the primary report and in the subreport `creditors`.

In addition, two different methods are used to conditionally execute subreports:

- The primary report calls JPL procedure `check_sum_bal`, which returns 1 or 0 to skip or invoke the subreport that follows. (Refer to Section 6.2.2 of the ReportWriter manual for a discussion of return codes from JPL procedures and C functions.)
- The subreport `creditors` calls JPL procedure `check_vendor_bal`, which resets the variable `subreports` or sets it to the subreport name `purchases`.

Figure 31 shows the screen layout for this example. The script and JPL procedures follow:

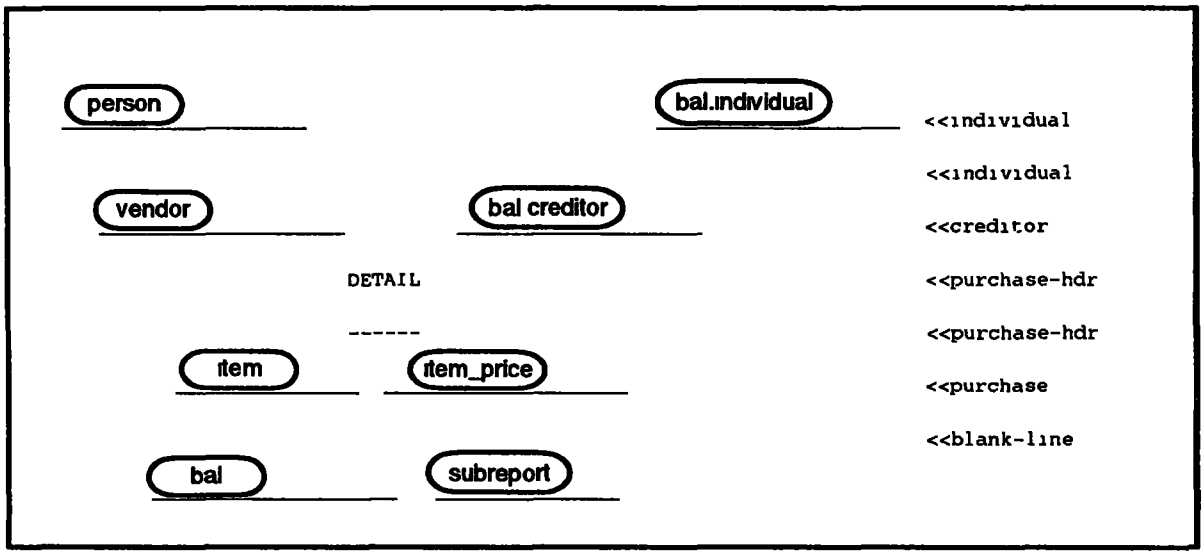


Figure 31: Subreport Example, Upgraded – Screen Layout (with field names)

```
# << BEGIN REPORT >>
# init
#        jpl = dbinit
# detail
#        query = 'select distinct person from all_bal'
#        jpl = get_sum_bal
#        area = individual
#        jpl = check_sum_bal
#        report = creditors
# insert
#        jpl = dbexit
# << END REPORT >>
#
# << BEGIN REPORT = creditors >>
# detail
#        query = 'select vendor, bal from all_bal \
#                where person = :+person and bal < 0'
#        area = creditor
#        jpl  = check_vendor_bal
#        report = :subreport
#        area = blank-line
# << END REPORT >>
#
# << BEGIN REPORT = purchases >>
# insert
#        area  = purchase-hdr
# detail
#        query = 'select item, item_price from stores \
#                where buyer = :+person and store = :+vendor'
#        area  = purchase
# << END REPORT >>


proc get_sum_bal
 sql select person, sum (bal) bal from all_bal where person =\
  :+person group by person
 if bal > 0
    cat bal "+" bal

proc check_sum_bal
 if bal >= 0
    return 1

proc check_vendor_bal
 if bal > -200
    cat subreport
 else
    cat subreport "purchases"
```

D.4
CALENDAR

Although producing a monthly calendar is an atypical use for the ReportWriter, the example presented in this section demonstrates a variety of techniques you will find applicable to more "traditional" reports.

In particular, the calendar requires that multiple rows from the database be presented across a single instance of a report area. The same requirement would apply if you were using ReportWriter to generate mailing labels, two or more across a page. The technique shown in this example avoids the use of onscreen arrays to receive the fetched data, as this approach would not give the developer the fine control over handling each row that is required for the application.

This report demonstrates the following techniques:

- Single instance of a report area to output multiple rows fetched from the database
- Lowest–level output in break footer (rather than in the `detail` statement)
- Computed break (field `weeks`)
- Order–sensitive processing (break header for `month`)
- Auxiliary `SELECT` using named cursor
- `breakcheck` needed between two `detail` JPL procedures
- Non–output variables in the report format screen
- Field name aliasing (`date.week2` and `text.week2`)
- Alternative output areas and suppressed output, controlled by a variable containing an area tag name (variable `weekarea` in the `week` break footer)
- Use of pre– and post–break values at break–footer time (variables `week` and `newweek`)
- Output area conditionally populated (field `divider` for "six–line" months)
- Arrays
- `shrink` keyword (footer area for break field `month`)

Sample output from this report appears in Figure 32. Two months are shown. Note that the August, 1992, calendar is a "six–line" month; this example compresses such calendars into five lines.

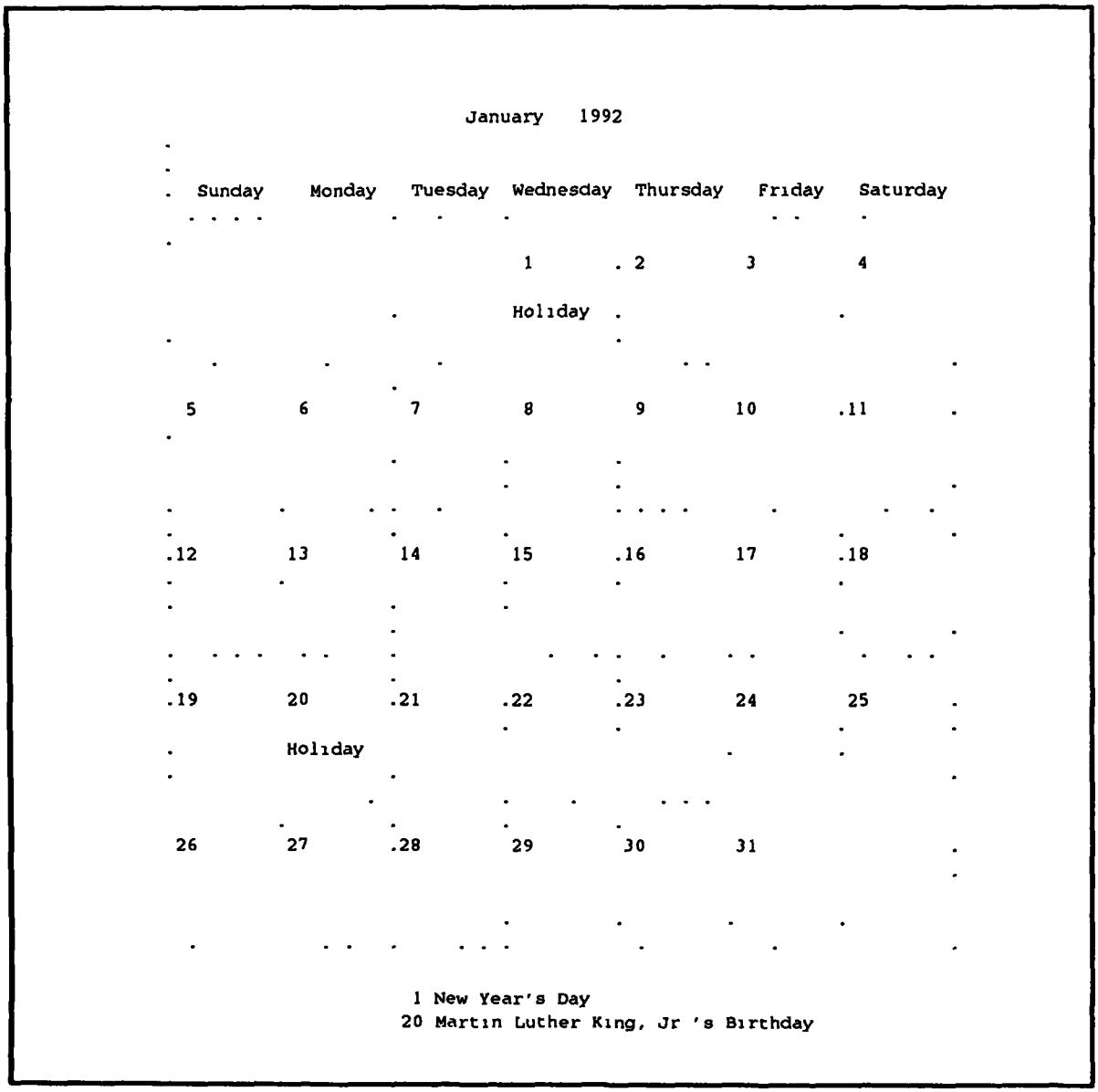


Figure 32: Calendar Report Sample Output

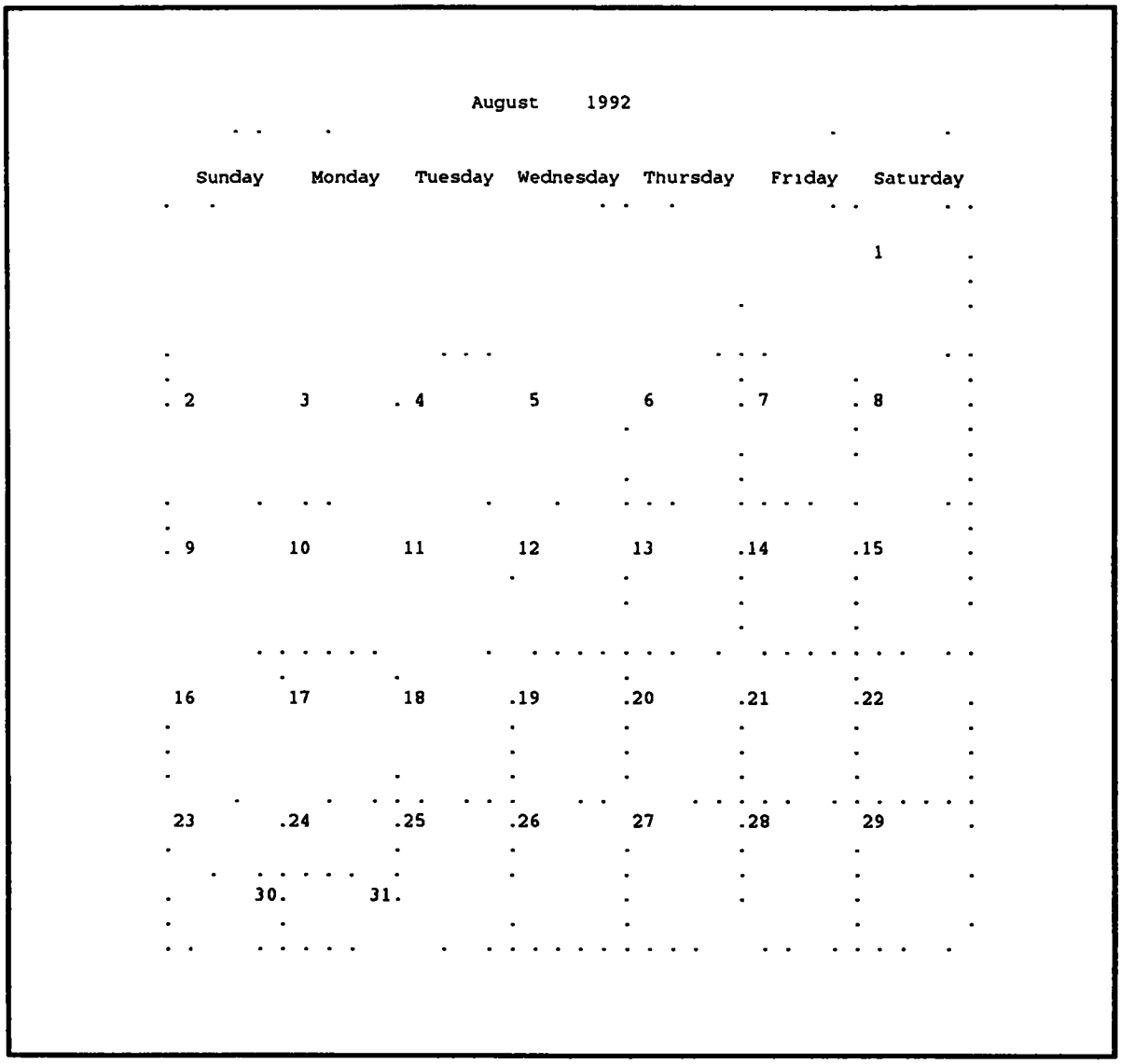


Figure 32 (cont'd.): Calendar Report Sample Output

The screen JPL module is reproduced below.

The first part of the JPL module contains the report script:

```
# << begin report >>
#
# /*
#
# This report produces a monthly calendar.  It reads from a table
# whose columns are as follows:
#
#    year       integer
#    month      integer
#    dayofmonth integer
#    dayofweek  integer
#    remark     char (40)
#
# In the table is one row for every day of the month, for as many
# months as are desired.
#
# An auxiliary table maps the month number to its name.  Its
# columns are
#
#    month      integer
#    monthname  char (9)
#
# */
#
# init jpl = on
#
# break field = year
#
# break field = month
#       header jpl = getmonthname
#              area = head
#       footer area = foot shrink
#       newpage
#
# break field = week
#       header jpl = clearweek
#       footer jpl = setweekarea
#              area = :weekarea
#
# detail query = "select * from calendar order by year, month,\
#                 dayofmonth"
#        jpl = computeweek
#        breakcheck
#        jpl = enterday
#
# << end report >>
```

The JPL procedures referenced in the report script follow in the JPL module:

```
proc getmonthname
#
# Get month name given its number, and clear holiday-listing
# arrays.
#
dbms declare c cursor for select monthname from months where\
     month = :month
dbms execute c
for nextholiday = 1 while nextholiday <= 10 step 1
{
        cat holdate[nextholiday]
        cat holname[nextholiday]
}
cat nextholiday "1"

proc clearweek
#
# Clear out the arrays to be used for the upcoming week.
#
vars i(10)
for i = 1 while i <= 7 step 1
{
        cat date[i]
        cat text[i]
}

proc setweekarea
#
# If this week is the 5th line of a 6-line month, move its data to
# the top arrays in the shared-week output area for output with the
# 6th line.  Otherwise output this week's data.  If this is a
# "6th-line" week, use the condensed output area, and add dots to
# split the second frame if needed.  ReportWriter has already stored
# the current week's data in the bottom arrays of the shared-week area
# because their names are aliases of the arrays in the single-week
# area.
#
# Because field "week" appears in a "break" statement in the report
# script, RW uses its pre-break value when this routine is executed
# (i.e., at break-footer time).  Field "newweek" does not appear in
# any break statement; therefore at break-footer time ReportWriter
# uses its post-break value.
#
vars i(10)
if week == 5 && newweek == 6
{
        cat weekarea
        for i = 1 while i <= 7 step 1
        {
```

```
                cat prevdate[1] date[i]
                cat prevtext[1] text[1]
        }
}
else if week == 6
{
        cat weekarea "week2"
        if date[2] == ""
                cat divider
        else
                cat divider ". . . ."
}
else
        cat weekarea "week"

proc computeweek
#
# Determine in which calendar line this day falls, and perserve its
# value in a variable not represented in a "break" statement, so that
# we may see the new value when the previous group's footer is
# computed.
#
math %t.0 week = (dayofmonth + 13 - dayofweek) / 7
cat newweek week

proc enterday
#
# Enter this day's data into the output arrays.
#
cat date[dayofweek] dayofmonth
if remark != ""
{
        cat text[dayofweek] "Holiday"
        cat holdate[nextholiday] dayofmonth
        cat holname[nextholiday] remark
        math %.0 nextholiday = nextholiday + 1
}
```

The screen layout for the calendar example is shown in Figure 33.

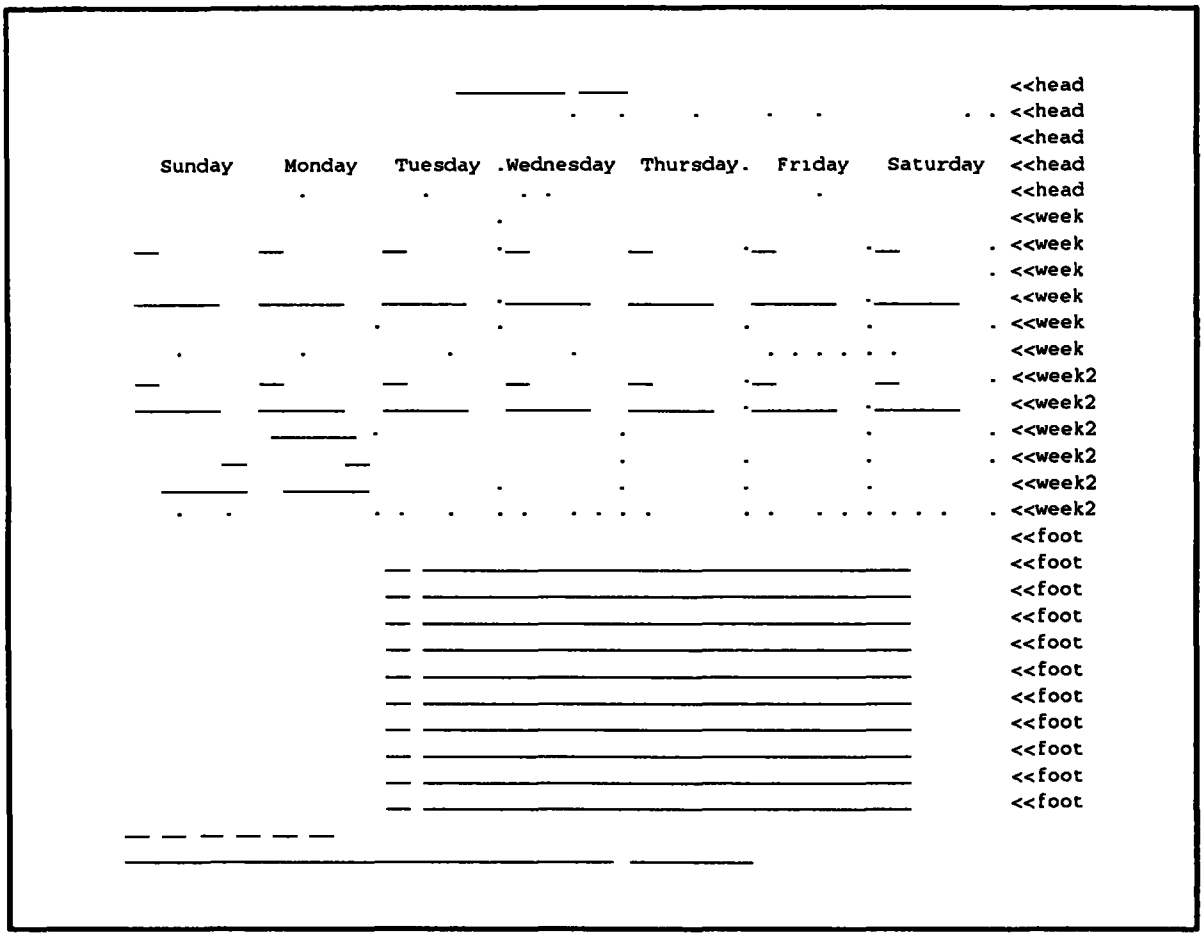


Figure 33: Calendar Report Screen Layout

Field descriptions, as generated by JAM's `f2asc` utility, are reproduced below:

```
F:monthname
#  NUMBER=1
   LINE=1 COLUMN=28
   UNFILTERED
   LENGTH=9 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:year
#  NUMBER=2
   LINE=1 COLUMN=38
   UNFILTERED
   LENGTH=4 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
```

```
F.date
#  NUMBERS=3, 4, 5, 6, 7, 8, 9
   LINE=7 COLUMN=2
#     (7,12 7,22 7,32 7,42 7,52 7,62)
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=7 HORIZ-DISTANCE=8
   WHITE UNDERLINE HILIGHT
   RIGHT-JUSTIFIED

F:text
#  NUMBERS=10, 11, 12, 13, 14, 15, 16
   LINE=9 COLUMN=2
#     (9,12 9,22 9,32 9,42 9,52 9,62)
   UNFILTERED
   LENGTH=7 ARRAY-SIZE=7 HORIZ-DISTANCE=3
   WHITE UNDERLINE HILIGHT

F:prevdate
#  NUMBERS=17, 18, 19, 20, 21, 22, 23
   LINE=12 COLUMN=2
#     (12,12 12,22 12,32 12,42 12,52 12,62)
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=7 HORIZ-DISTANCE=8
   WHITE UNDERLINE HILIGHT
   RIGHT-JUSTIFIED

F:prevtext
#  NUMBERS=24, 25, 26, 27, 28, 29, 30
   LINE=13 COLUMN=2
#     (13,12 13,22 13,32 13,42 13,52 13,62)
   UNFILTERED
   LENGTH=7 ARRAY-SIZE=7 HORIZ-DISTANCE=3
   WHITE UNDERLINE HILIGHT

F:divider
#  NUMBER=31
   LINE=14 COLUMN=13
   UNFILTERED
   LENGTH=7 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:date.week2
#  NUMBERS=32, 33
   LINE=15 COLUMN=9
#     (15,19)
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=2 HORIZ-DISTANCE=8
   WHITE UNDERLINE HILIGHT
   RIGHT-JUSTIFIED
```

```
F:text.week2
#  NUMBERS=34, 35
   LINE=16 COLUMN=4
#    (16,14)
   UNFILTERED
   LENGTH=7 ARRAY-SIZE=2 HORIZ-DISTANCE=3
   WHITE UNDERLINE HILIGHT

F:holdate
#  NUMBERS=36, 38, 40, 42, 44, 46, 48, 50, 52, 54
   LINE=19 COLUMN=22
#    (20,22 21,22 22,22 23,22 24,22 25,22 26,22 27,22 28,22)
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=10 VERT-DISTANCE=1
   WHITE UNDERLINE HILIGHT
   RIGHT-JUSTIFIED

F:holname
#  NUMBERS=37, 39, 41, 43, 45, 47, 49, 51, 53, 55
   LINE=19 COLUMN=25
#    (20,25 21,25 22,25 23,25 24,25 25,25 26,25 27,25 28,25)
   UNFILTERED
   LENGTH=40 ARRAY-SIZE=10 VERT-DISTANCE=1
   WHITE UNDERLINE HILIGHT

F:month
#  NUMBER=56
   LINE=29 COLUMN=1
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:week
#  NUMBER=57
   LINE=29 COLUMN=4
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
   RIGHT-JUSTIFIED

F:newweek
#  NUMBER=58
   LINE=29 COLUMN=7
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
   RIGHT-JUSTIFIED

F:dayofmonth
#  NUMBER=59
   LINE=29 COLUMN=10
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
```

```
F:dayofweek
#  NUMBER=60
   LINE=29 COLUMN=13
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:nextholiday
#  NUMBER=61
   LINE=29 COLUMN=16
   UNFILTERED
   LENGTH=2 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:remark
#  NUMBER=62
   LINE=30 COLUMN=1
   UNFILTERED
   LENGTH=40 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT

F:weekarea
#  NUMBER=63
   LINE=30 COLUMN=42
   UNFILTERED
   LENGTH=10 ARRAY-SIZE=1
   WHITE UNDERLINE HILIGHT
```

INDEX

Symbols

A

B

C

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

JAM/Presentation

interface

for

OSF/Motif,

OPEN LOOK

and

MS Windows

Release 1.4

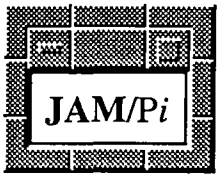
This is the manual for the **JAM**/Presentation *interface* for Microsoft Windows, OSF/Motif, and OPEN LOOK. It is as accurate as possible at this time; however, both this manual and **JAM**/Presentation *interface* itself are subject to revision.

Please 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



Chapter 1
Introduction

1.1
ABOUT THIS DOCUMENT

This document is intended to introduce the **JAM/Presentation** *interface* to developers who are *already* familiar with **JAM®**. It is *not* intended as a substitute for any part of Volumes I and II of the **JAM** manual. If you are new to **JAM**, please read the **JAM** manual first.

Conceptually, this manual is separated into two parts. The first part describes what the **JAM/Presentation** *interface* is and explains how to use it. Chapters 1 through 5 comprise this part. The balance of the manual is a reference, giving the details of the various features and functions in the product. An appendix at the end of the manual contains a glossary of terms associated with Graphical User Interfaces (GUI's) and **JAM**. These terms are used throughout the manual. Please refer to Appendix A if you are confused about the meaning of any terms used.

1.1.1
Conventions

All conventions in the **JAM** manual are adopted for this manual. In addition, the following icons indicate that a particular section applies to one presentation interface only.

Text in the shaded area after a W icon refers only to the **JAM/Presentation** *interface* for Windows.

Text in the shaded area after an M icon refers only to the **JAM/Presentation** *interface* for Motif.

O Text in the shaded area after an O icon refers only to the JAM/Presentation *interface* for OPEN LOOK.

1.2
WHAT IS THE JAM/Presentation *interface*?

The **JAM**/Presentation *interface* (**JAM**/P*i*) product line provides a layer between the user and the application that enables **JAM** to support a variety of textual and graphical environments. **JAM**/P*i* products include:

- **JAM**/Presentation *interface* for Microsoft Windows (P*i*/Windows)
- **JAM**/Presentation *interface* for Motif (P*i*/Motif)
- **JAM**/Presentation *interface* for OPEN LOOK (P*i*/OPEN LOOK)
- **JAM**/Presentation *interface* for Graphics (P*i*/Graphics)

Presentation interfaces for other environments, such as Macintosh, are in development.

Traditional, character–based **JAM**, is referred to in this document as "character **JAM**".

This document covers the **JAM**/Presentation *interface* for three Graphical User Interfaces (or GUI's): Microsoft Windows, Motif and OPEN LOOK. P*i*/Graphics is covered in a separate document. The abbreviation **JAM**/P*i*, when used here, encompasses P*i*/Windows, P*i*/Motif and P*i*/OPEN LOOK, but not P*i*/Graphics.

The **JAM**/P*i* layer transforms **JAM** into a GUI compliant product. JYACC's philosophy is that **JAM** should be a flexible tool for creating device independent software applications. Figure 1 illustrates this layered concept.

JAM/P*i* retains **JAM** functionality but adopts the look and feel of the presentation device. Preserving the look and feel of the GUI was the overriding concern in the development of **JAM**/P*i*.

The previous paragraph should not be taken to imply that **JAM**/P*i* applications only *look like* GUI applications. In fact, applications developed with **JAM**/P*i* *are* GUI *compliant* applications.

1.3
USING JAM/P*i* EFFECTIVELY

In order to effectively use **JAM**/P*i*, you must have an understanding of **JAM**. **JAM** screens are built from **JAM** objects: fields, groups, menus and display text. **JAM** ap-

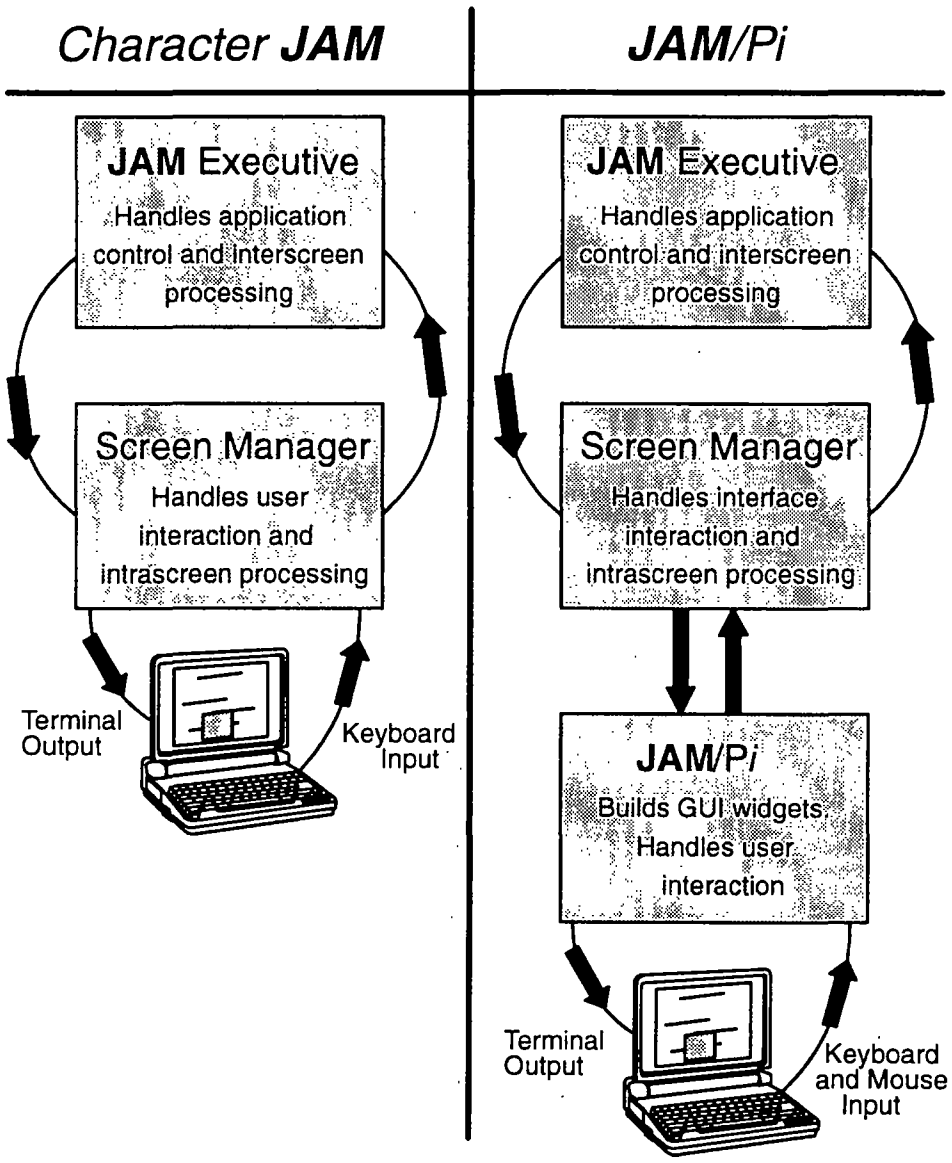


Figure 1: Schematic models of character **JAM** and **JAM**/P*i*. User input, terminal output, and screen appearance are handled by the Presentation *interface* layer instead of the Screen Manager.

plications are built from **JAM** screens. The Screen Manager handles processing within a screen, and the **JAM** Executive provides interscreen links and data flow control.

JAM/Pi provides a link to the GUI world by converting **JAM** objects into GUI widgets. But **JAM/Pi** provides a higher level interface than that available from most products. For example, with **JAM/Pi** the developer has no need to worry about callbacks for each widget on a screen. The **JAM** Screen Manager deals with these issues. Similarly, interscreen links are easily specified in **JAM/Pi**, and the developer does not need to define what happens, for example, when the close button on a screen is pressed by the user. These events are handled by the **JAM** Executive, and may be defined on an application–wide basis.

The best way to use this product is to develop screens from a functional viewpoint, and worry about their appearance as an implementation detail. Don't take the approach that you want a certain six widgets on a screen and then go about placing them there. The best approach is to design screens with **JAM** objects and **JAM** interactions in mind. Once a screen has been created, you can worry about changing the type of widget used in a particular case. **JAM** provides a default transformation of each type of **JAM** object into a GUI widget, but the developer is free to override the default choices

1.4
OVERVIEW OF FEATURES IN JAM/Pi

1.4.1
Portability Across Environments

Applications developed in character **JAM** can be run without modification under Pi/Windows, Pi/Motif or Pi/OPEN LOOK. **JAM** screens adopt the look and feel of the GUI, but **JAM** functionality remains constant. **JAM** screen binaries are identical among environments. Each environment simply interprets them in its own way.

In many real world applications the developer will wish to make certain cosmetic modifications to screens in order to take maximum advantage of GUI features. Most of these modifications are portable back to character **JAM**, as well as to other Presentation *interfaces*.

Certain features in **JAM/Pi** are extensions to **JAM**, and are not currently portable back to character–based environments. These features are implemented so they translate to parallels in character **JAM**. For example, menu bars translate to keysets. Planned enhancements to character **JAM** will eliminate many of these limitations.

1.4.2

Compatibility with Character JAM

From the developer's point of view, the functionality of **JAM/P*i*** is virtually identical to character **JAM**. The Screen Editor, Data Dictionary Editor, and Keyset Editors retain their functionality, as does Application Mode within the Authoring tool. Navigational techniques and mouse behavior differ slightly among interfaces, but conceptually the **JAM** authoring tools work as they always have.

From the end–user's point of view on the other hand, **JAM/P*i*** applications are purely GUI based.

1.4.3

Support for GUI features

In order to create real GUI applications, **JAM/P*i*** provides support for a wide range of GUI features.

Transformation of Objects and Text

Each type of object on a **JAM** screen is transformed into an equivalent GUI object. For example, in Figures 2 and 3 we see a **JAM** menu, a data entry field, a checklist group, and display text in character **JAM** and in P*i*/Motif respectively.

Each **JAM** window comes up as its own GUI window, with appropriate decorations as prescribed by the window manager. These windows can be moved, resized, scrolled, and in some cases, iconified.

Extended Functionality

Another example of GUI feature support is the implementation of menu bars, which are often the primary tool for user interaction in GUI applications. The keyset hook in character **JAM** may be used in **JAM/P*i*** to enable menu bars. Like keysets, menu bars are created as external components to an application, and accessed from disk files, libraries, or as memory resident 'files'. This architecture minimizes the steps required to convert applications from one environment to another. For applications that are already using keysets, a utility is provided for converting keysets into menu bars.

Figures 4 and 5 compare two applications. In the first, keysets are used to navigate. In the second, the keysets have been converted into menu bars.

Extended Fonts and Colors

GUI's offer a host of extended font and color choices that are unavailable on most character–based platforms. In order to support these enhancements and maintain portability,

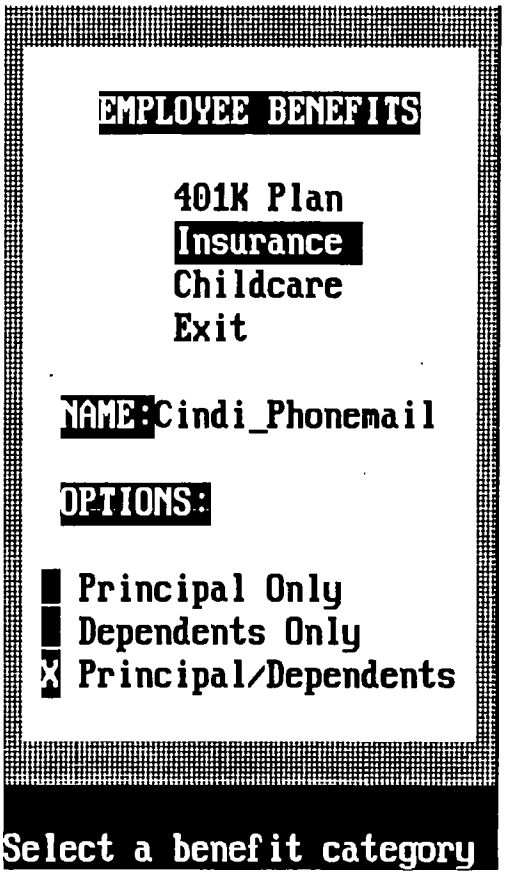


Figure 2: Screen in character **JAM**.

cosmetic screen alterations taking advantage of these extended display options are indicated by special comments in the JPL modules associated with each field and screen. These comments are called extensions. The following can all be specified as extensions: font, widget size, widget position and alignment, specialized widgets, extended colors, title bars, bitmaps, border decorations, and graphics. Formatted screens are provided to aid the developer in entering extensions.

Since extensions are stored in JPL comments, they are portable. In environments such as character mode, where extensions are unavailable, the comments are simply ignored.

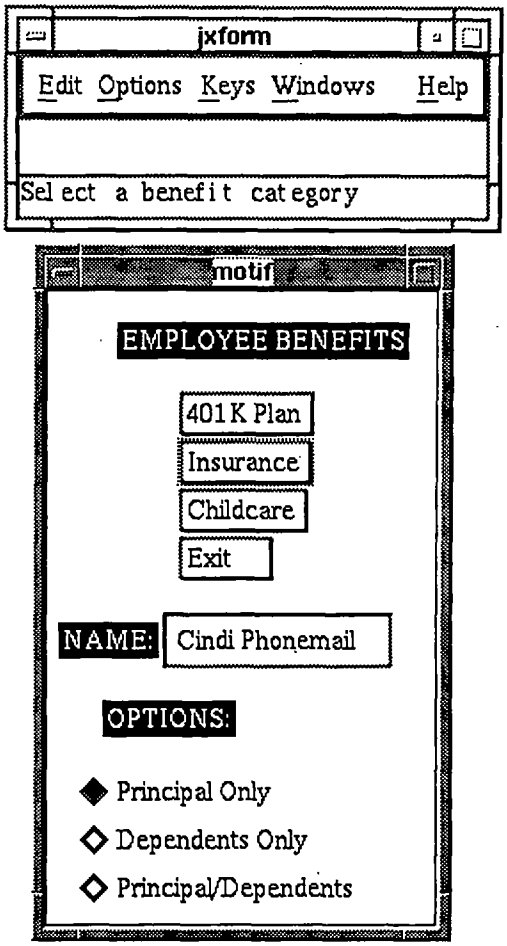


Figure 3: Same screen in P*i*/Motif.

Application Defaults

Resource files and initialization files provide for customization on a screen–wide and application–wide basis. These are external to **JAM**, and therefore may be changed by the end–user. Resource files determine the display characteristics and user interface behavior of an application. Items such as default colors, default fonts, border and shadow characteristics, and keyboard focus policy can all be included if the GUI supports them.

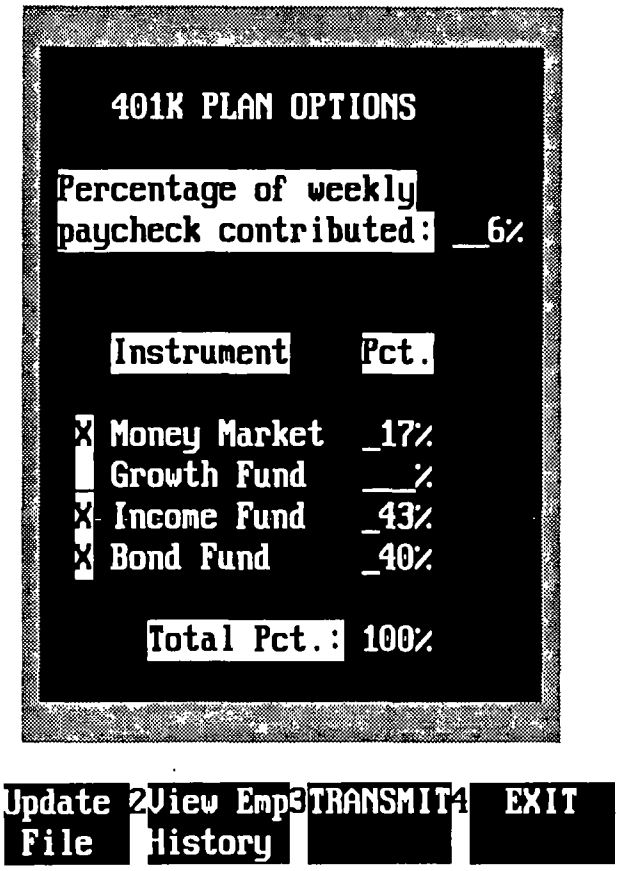


Figure 4: Character–based screen with keysets

The structure and contents of resource and initialization files are specific to the GUI being employed.

Extended Library Routines

JAM/Pi also provides extended library routines for functionality specific to GUI's. For example, routines are available to modify menu bars at runtime and interact with the GUI directly. While some of these extensions are not portable among environments, they provide additional features in situations where portability is not an issue.

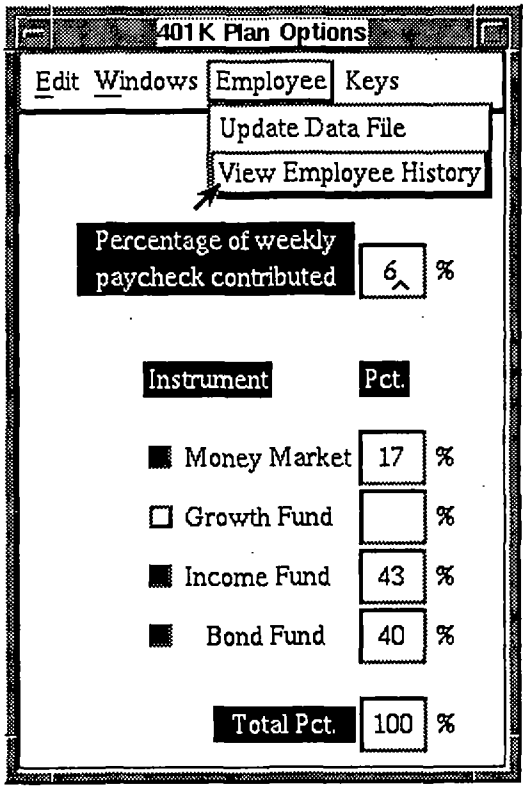
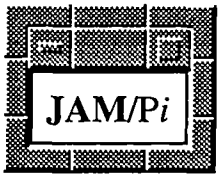


Figure 5: P*i*/Motif Screen with menu bars



Chapter 2
JAM Objects into GUI Widgets

This chapter examines how **JAM** screen objects are transformed into GUI widgets under **JAM/P*i***. An illustration of each widget is provided, along with a brief description of how the user interacts with it.

2.1
INTRODUCTION

GUI screens are composed of widgets (also called controls in MS Windows). When a **JAM** screen is brought up under **JAM/P*i***, **JAM** screen objects become widgets. Each type of **JAM** object is transformed into a particular type of widget. Each **JAM** object has a default transformation, but you may choose to use a different widget than the default. The table below lists the default transformations using Motif terminology. Names for all the widgets in the various interfaces are listed in Chapter 7.

JAM Object	*Default Widget*
Display Text	Label Widget
Data Entry Field	Text Widget
Protected Field	Label Widget
Menu	Push Button
Radio Button Group	Radio Toggle Buttons
Checklist Group	Checklist Toggle Buttons
Border	– none –
Line Drawings	– none –

Additional widgets that a developer can specify are listed below. The specifics of how to create each widget are detailed in Chapters 5 and 6.

- List box
- Optionmenu (or combo box)
- Multiline text widget
- Multiline button
- Scale widget
- Pixmap

There are three additional widgets used for screen decoration. They are:

- Separator (horizontal or vertical line)
- Frame
- Box

2.2 WIDGET ATTRIBUTES

Before going into the specifics of how **JAM** objects are transformed into widgets, it is important to understand where widgets get their attributes from.

2.2.1 Widget Attribute Hierarchy

The design of each widget is determined by the GUI, but various attributes may be set by the developer. Certain attributes, such as foreground and background colors, are inhereted from **JAM**. **JAM/Pi** extensions may be used to override these inhereted attributes. Other attributes, such as font, may be set on an application–wide, screen–wide, or individual widget basis.

JAM/Pi provides a hierarchical system for determining attributes. It goes from GUI defaults files for application–wide settings, to screen extensions for screen–wide settings, to **JAM** field attributes and field extensions for widget–specific settings. Figure 6 illustrates the hierarchy that determines which attributes are effective for a widget. The various ways of setting attributes are summarized below.

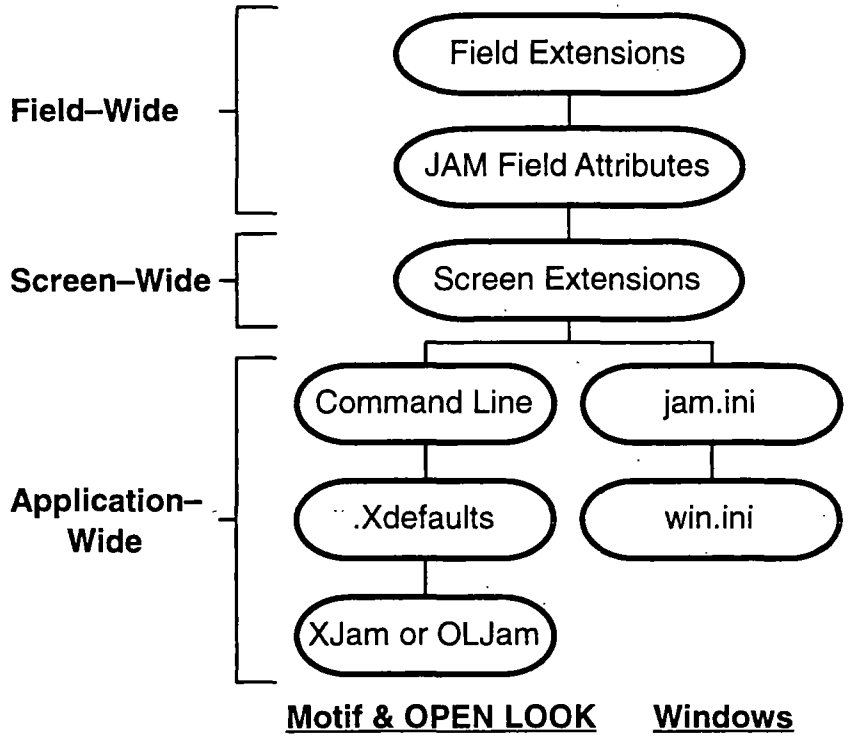


Figure 6: Hierarchy for widget attributes. Field extensions override screen extensions, which override the command line, etc.

2.2.2

Application–Wide Attributes

Application–wide attributes are set in GUI defaults files. These are external to a **JAM** application. Their structure is determined by the GUI. Note that the end user may edit these files, thereby changing the default values. Application–wide attributes may be set in the following locations:

W

`win.ini` file (or Windows Control Panel)

Here you may alter system–wide colors for the various components of the Windows display. Either use the Color tool available on the Windows Control Panel, or edit the `win.ini` text file directly. The `win.ini` file affects the entire Windows environment.

application initialization file

Distributed as `jam.ini`. Here you may specify the application default font, the values for **JAM** colors, a set of GUI–independent font and color names, and certain behavioral characteristics. See Chapter 7 for details.

M

application resource file

Distributed as `XJam`. This is the application specific resource file for P*i*/Motif. It is normally found in the application defaults directory (usually `/usr/lib/X11/app-defaults`). Here you may specify default values for widget colors, fonts and other attributes. Attributes may be specified for the whole application, for the widgets on a particular screen, for a class of widgets, or for a specific named widget. Attribute specifications that are not application–wide override screen–wide and field–wide attributes.

The mapping between **JAM** colors and Motif colors, as well as a set of GUI independent font and color names may also be specified here. Resource files are discussed in detail in Chapter 7.

`.Xdefaults` file

This is the user specific resource file in the X Window System. It is normally found in the user's home directory. All settings that can be made in the application resource file may also be made here. Settings in this file override those in the application resource file for the particular user.

command line

Here you may specify a default font and certain options relating to P*i*/Motif behavior. These settings override resource files. See section 7.7.1.

O

application resource file

Distributed as `OLJam`. This is the application specific resource file for P*i*/OPEN LOOK. It is normally found in the application defaults directory (usually `$OPENWINHOME/lib/app-defaults`). Here you may specify default values for widget colors, fonts and other attributes. Attributes may be specified for the whole application, for the widgets on a particular screen, for a class of widgets, or for a specific named widget. Attribute specifications that are not application–wide override screen–wide and field–wide attributes.

The mapping between **JAM** colors and OPEN LOOK colors, as well as a set of GUI independent font and color names may also be specified here. Resource files are discussed in detail in Chapter 7.

`.Xdefaults` file

This is the user specific resource file in the X Window System. It is normally found in the user's home directory. All settings that can be made in the application resource file may also be made here. Settings in this file override those in the application resource file for the particular user.

command line

Here you may specify a default font and certain options relating to P*i*/OPEN LOOK behavior. These settings override resource files. See section 7.7.1.

2.2.3

Screen–Wide Attributes

Screen–wide attributes may be set via the:

- screen extensions

  These are used to specify a default background color, foreground color and font for widgets on the screen. Screen extensions are stored in the screen–level JPL comments and may entered through special formatted screens accessed via SPF11. Screen extensions are detailed in Chapters 5 and 6.

M **O** In P*i*/Motif and P*i*/OPEN LOOK, attributes specified in the resource file that refer to a screen name are equivalent to screen extensions and override them.

2.2.4

Widget–Specific Attributes

Widget–specific attributes may be set through the:

- **JAM** display attributes window

  Here you may specify attributes for individual fields or groups. Certain settings, such as blinking, may not be implemented in certain interfaces. This window is accessed via PF4 in the Screen Editor.

- field extensions

  Attributes set here override all other settings. Attributes that may be set include: widget size, font, extended foreground and background colors, incremental positioning, and specialized widgets. Field extensions are stored in the field–level JPL comments and may be entered through special formatted screens accessed via SPF12. For details see Chapters 5 and 6.

In P*i*/Motif and P*i*/OPEN LOOK, attributes specified in the resource file that refer to a widget or widget class are equivalent to field extensions and override them.

2.3
TRANSFORMATION INTO WIDGETS

The following sections detail the transformation of each **JAM** object into its GUI counterpart.

2.3.1
Display Text and Protected Fields

Regions of display text become label widgets in **JAM/P*i***. Regions of display text are not fields, and therefore cannnot have field extensions. They do however have **JAM** display attributes, and can inheret other attributes from the screen.

Fields protected from data entry and tabbing also become label widgets. They have an advantage over display text in that they can have field extensions, making them more flexible. For example, if you wish to change the font of a single region of display text, convert it into a protected field and change the font with a field extension. Using protected fields also allows label widgets to be right justified. Right justified label widgets are discussed further in section 3.2.1, in relation to positioning.

Figure 7 illustrates how label widgets appear in Motif and Windows.

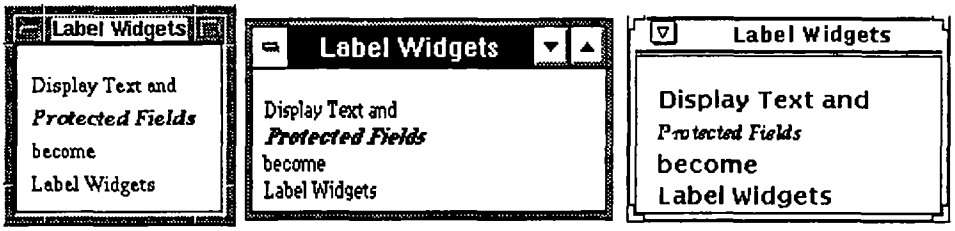


Figure 7: Label widgets in P*i*/Motif, P*i*/Windows and P*i*/OPEN LOOK.

2.3.2
Data Entry Fields

Data entry fields become text widgets in **JAM/P*i***. The look and feel of the text widget is determined by the GUI, but the **JAM** field edits control its behavior.

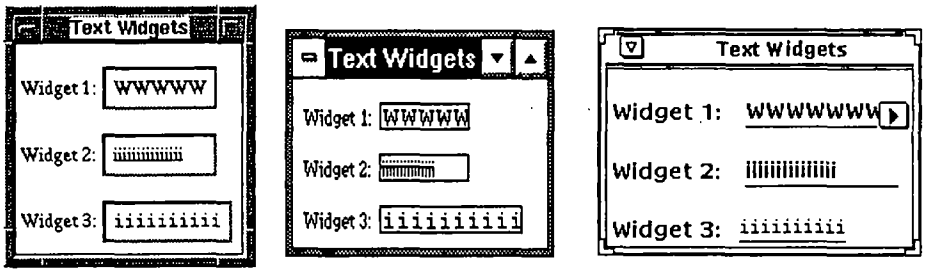


Figure 8: Text widgets in Pi/Motif, Pi/Windows and Pi/OPEN LOOK.

2.3.3

Arrays

By default, each array element is a separate text widget. Field extensions provide ways to change arrays into multiline text widgets (for data entry fields) or list boxes (for selection fields). There is also a field extension to assure that individual array elements are spaced evenly on the screen. Refer to Chapters 5 and 6 for details.

Arrays protected from data entry and tabbing become label widgets.

An array may be scrolled by dragging the mouse cursor beyond the edge of the array in the direction you wish to scroll, or by using the keyboard or scrolling indicators (if present). List boxes and multiline text widgets may be scrolled and shifted from optional scroll bars.

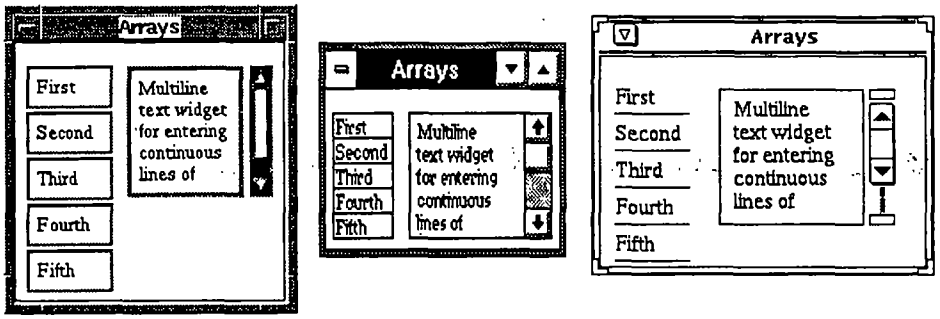


Figure 9: An array and multiline text widget in Pi/Motif, Pi/Windows and Pi/OPEN LOOK.

2.3.4

Menus

Menu fields appear as push buttons in JAM/P*i*. Push buttons perform an action when activated with the mouse or keyboard. Label text is centered within the push button widget, and drop shadows make the widget appear to protrude from the screen.

As in character JAM, menu fields must have the menu edit and be protected from data entry and tabbing in order to look and act as menus in both data entry and menu modes.

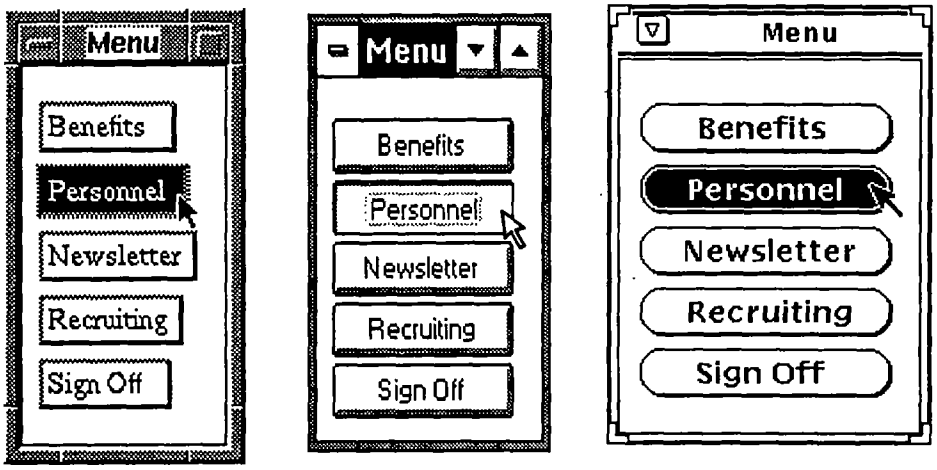


Figure 10: A set of menu fields in P*i*/Motif, P*i*/Windows and P*i*/OPEN LOOK. The "Personnel" option is selected.

NOTE: The color of regular push buttons in Windows cannot be changed by JAM. Windows enforces a single color for all push buttons. This color may be set on a system-wide basis through the Windows control panel.

JAM/P*i* does however provide a field extension to create a multiline push button. Multiline buttons can have custom colors. If you wish to change the color of a regular push button, make it a multiline button with only one line. See Chapters 5 and 6 for details.

M In Pi/Motif, the alignment of text in buttons and labels is controlled by the `alignment` resource, as in:

```
XJam*XmPushButton*alignment:  ALIGNMENT_BEGINNING
```

You may change the value for this resource to `ALIGNMENT_CENTER` for center justification, or `ALIGNMENT_END` for right justification. For more information on resources, refer to Chapter 7.

O In Pi/OPEN LOOK, the alignment of text in buttons is controlled by the `labelJustify` resource, as in:

```
OLJam*areaOblongButton.labelJustify:  center
```

You may change the value for this resource to `left` for left justification. Right justification is not supported. For more information on resources, refer to Chapter 7.

Menu bars are also available in **JAM/Pi**. Refer to Chapter 8.

2.3.5

Groups

Groups become sets of toggle button widgets in **JAM/Pi**. Radio buttons have one style and checklists another. The details are set by the GUI. The checkbox on a toggle button is filled in when the entry is selected, and empty when it is unselected.

A group can be converted into a list box widget via the field extensions. List boxes are appropriate for groups since groups are selection criteria, rather than data entry fields.

M In Pi/Motif, groups without the checkbox edit appear as a toggle buttons without a checkbox. In this form they look like push buttons that remain pushed in after being pressed.

Alignment of text in a toggle button is controlled by the `alignment` resource, as in:

```
XJam*XmToggleButton*alignment:  ALIGNMENT_BEGINNING
```

You may change the value for this resource to `ALIGNMENT_CENTER` for center justification, or `ALIGNMENT_END` for right justification. For more information on resources, refer to Chapter 7.

O In P*i*/OPEN LOOK, the alignment of text in toggle buttons is controlled by the `labelJustify` resource, as in:

```
OLJam*area.oblongButton.labelJustify: center
```

You may change the value for this resource to `left` for left justification. For more information on resources, refer to Chapter 7.

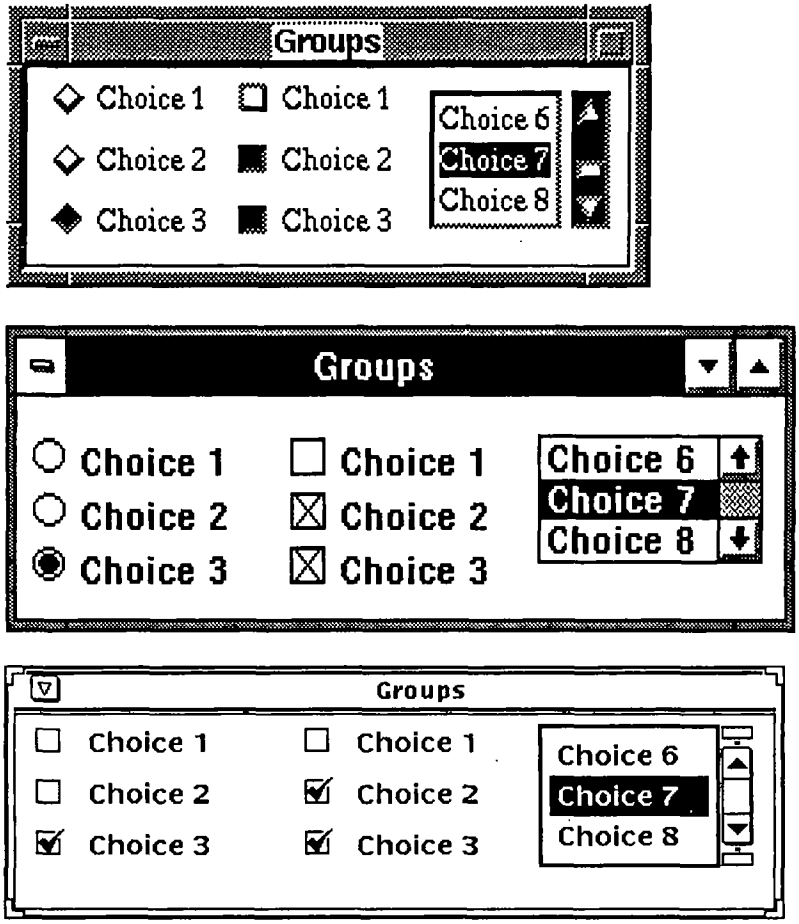
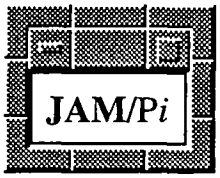


Figure 11: A radio button, checklist, and list box in P*i*/Motif, P*i*/Windows and P*i*/OPEN LOOK.



Chapter 3
Arranging Screens in JAM/Pi

When JAM screens are displayed in JAM/Pi, JAM objects are transformed into widgets. The size of a widget may be different than the size of the JAM object that it replaces. In fact, most widgets are slightly larger than their character based counterparts. In order to convert JAM screens into GUI screens without enlarging them excessively, JAM/Pi uses a positioning algorithm that attempts to fit widgets onto screens with as little disturbance as possible to the relative alignment of the objects.

3.1
OVERVIEW OF POSITIONING

Each JAM screen has a grid of rectangular cells whose default size is determined by the font in use. The display text and fields that are the basic building blocks of JAM screens are created in draw mode by typing text or underscores. Each character or underscore in character JAM occupies one grid cell, and every grid cell is the same size. This is true in character JAM and in *draw mode* of JAM/Pi.

In *test and application modes* of JAM/Pi though, fields and display text are converted into widgets. For example, data entry fields become text widgets; menu fields become push button widgets; and display text and protected fields become label widgets. GUI widgets may or may not fit into the cells that they were created in, in draw mode.

When a realized widget is larger than the cells it was drawn in, JAM/Pi stretches some of the rows or columns of the grid to accommodate the widget. This means that grid cells in test and application modes of JAM/Pi are *not all the same size.*

The grid in JAM/Pi is elastic; its size depends upon the objects on the screen. JAM/Pi stretches the grid only as much as is necessary. In fact, if whitespace is available to the right of a left justified widget or to the left of a right justified widget, JAM/Pi uses up that space before it stretches the grid. When the grid stretches, cells don't stretch indi-

vidually. Rather, entire rows or columns of cells stretch, assuring that other objects on the screen remain properly aligned. Figure 12 illustrates the elastic grid.

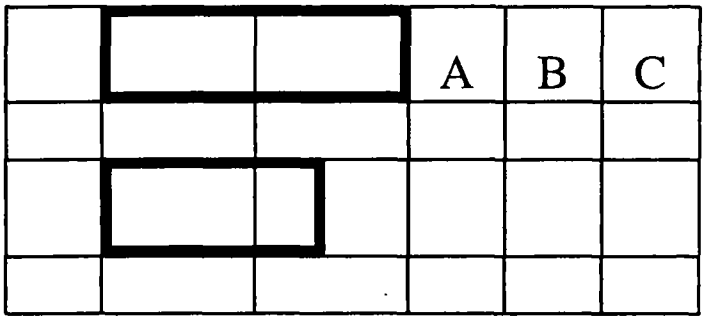


Figure 12: A schematic illustrating the elastic grid. The second and third columns have stretched, as have the first and third rows.

Although the grid stretches to accommodate large widgets, it does not shrink to accommodate small widgets. When a widget is smaller than the cells that it was drawn in, it anchors to a particular cell, and occupies only part of the available space. A widget anchors based on its justification: right justified widgets anchor by default on their right; left justified widgets anchor by default on their left.

For example, the widget in row 3 of Figure 12 is left justified. It anchors on its left.

The positioning algorithm is designed to allow maximum portability between character mode and GUIs. It maintains widget alignment even when the font or size of a widget changes. The following rule of thumb applies to positioning:

- Left justified fields that begin in the same column result in left aligned widgets.
- Right justified fields that end in the same column result in right aligned widgets.

Figures 13 and 14 compare a screen in draw mode and test mode of Pi/Motif.

Note that **JAM** objects appear as widgets only in test and application modes, not in draw mode.

In Pi/Windows, fields appear in boxes in draw mode.

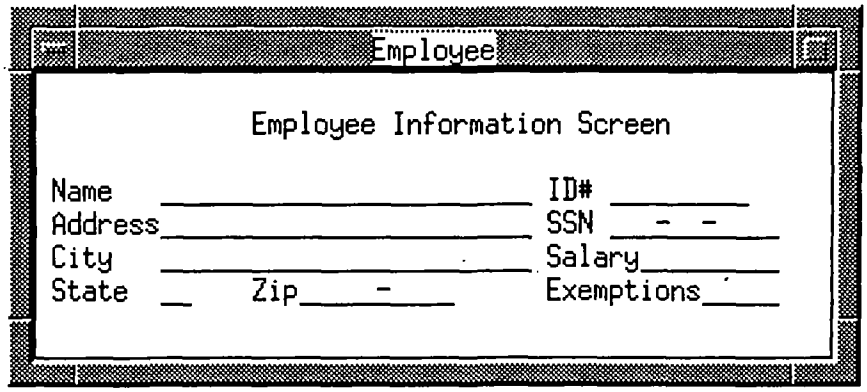


Figure 13: A **JAM** screen in draw mode of P*i*/Motif.

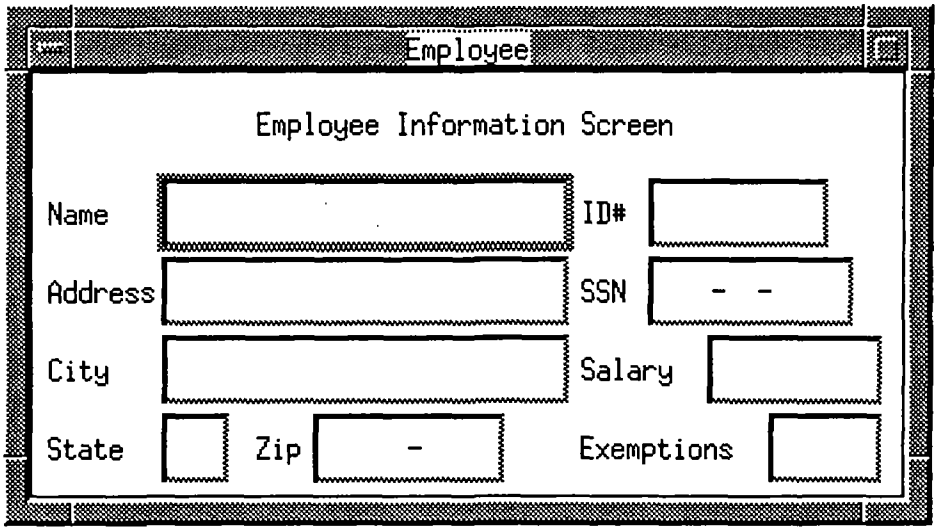


Figure 14: The same **JAM** screen in test mode. Draw mode looks like character **JAM**, while test mode looks like a GUI screen.

Notice how the Name, Address and City text widgets in Figure 14 stretch the grid horizontally, pushing the other objects on the screen to the right. Vertically, the last four rows stretch to accommodate the text widgets in them. As the grid stretches, the GUI window containing the **JAM** screen expands to accommodate it.

3.2 ANCHORING

In Figure 14, the ID# and SSN fields align on their left side in test mode, because they are left justified fields. The Salary and Exemptions fields align instead on their right side, because they are right justified. The alignment differences are due to where the widgets are anchored. Anchoring comes into play when a widget is not the same size as the cells allotted to it.

3.2.1 Anchoring by Field Justification

Each widget is anchored to a specific cell in the grid. The default anchor point of a widget is based on its justification. Right justified widgets anchor by default on their right: to the last (or rightmost) cell in which they are drawn. All other widgets anchor by default on their left: to the first (or leftmost) cell in which they are drawn. When the grid expands, widgets maintain their anchor points, and move along with the expanded grid. Widgets don't expand to fit the grid, rather the grid expands, if necessary, to fit the widgets.

Using field justification to determine alignment ensures compatibility with character **JAM**. For example, a column of numbers in right justified fields that line up on their right in character **JAM** will also line up on their right in **JAM/P***i*. A set of left justified data entry fields that start in the same column in **JAM** will maintain their left alignment in **JAM/P***i*, regardless of how the grid expands.

Alignment follows justification by default. If you wish to change the anchor point of a widget, use the `halign` or `valign` field extensions. These are described below.

3.2.2 Horizontal Anchoring: the `halign` Field Extension

The default positioning behavior specifies the anchor points of objects based on their field justification. The `halign` field extension (pronounced "aitch – align") overrides the default anchoring. Field extensions are documented in Chapters 5 and 6.

`halign` takes one argument, which is a number between zero and one. An `halign` of zero means that the left edge of the widget should anchor in its first (or leftmost) cell.

Zero is the default `halign` for left justified fields. An `halign` of one means that the right edge of the widget should anchor in its last (or rightmost) cell. This is the default for right justified fields. An `halign` between zero and one means that the widget should anchor proportionally between its first and last cells. Thus, an `halign` of `.5` means that the center of the widget should anchor in the center of the available cells.

The schematic diagram below represents a screen containing three text widgets of length 3 which span columns that have been stretched by a large label widget.

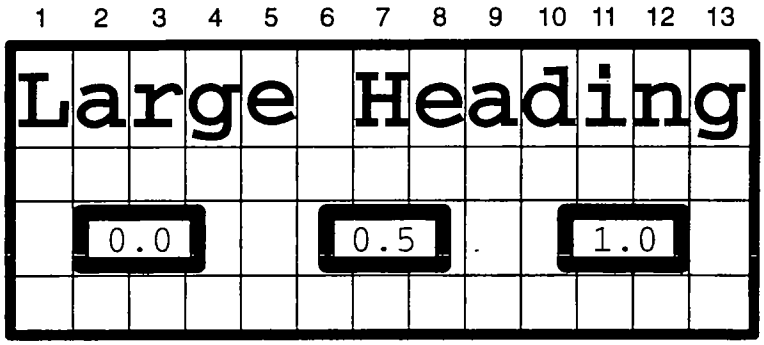


Figure 15: A screen containing a large heading and three data entry fields of length 3. The fields start in columns 2, 6 and 10, respectively. The `halign` of each field is shown as the field's contents.

In Figure 15, the large heading that runs the length of the screen stretches the grid. Each widget below is thus smaller than the cells available for it (3 columns worth of cells). `halign` determines where within its allotted cells a widget anchors.

Note that `halign` only has an effect when a widget is larger or smaller than its available cells.

3.2.3
Vertical Anchoring: the `valign` Field Extension

By default, all objects align vertically in the center of their row or rows. The `valign` field extension (pronounced "Vee – align") specifies some other alignment. Like `halign`, `valign` takes one argument, a number between zero and one. Zero indicates that the top of the widget should align with the top of the top cell. One indicates that the bottom of the widget should align with the bottom of the bottom cell. Decimal values

in between indicate proportional alignment between the top and bottom cells. The default `valign` for all objects is `.5`, indicating center alignment.

3.2.4

Anchoring Display Text

Regions of display text become left justified label widgets in **JAM/*Pi***. Left justified widgets have a default `halign` of `0`, and thus anchor on their left. Regions of display text are not fields, and therefore cannot be right justified or have field extensions. To change the alignment of a region of display text, you must convert the text into a protected field. Fields protected from data entry and tabbing also become label widgets in **JAM/*Pi***, but they have an advantage over display text in that they can be right justified and have field extensions. This means that their alignment can be adjusted. It also allows a label widget to have a font other than the default screen font.

A case where you might wish to anchor text on the right is in a field label. Field labels should retain their relationship to a field, regardless of the font used or how the grid stretches. By converting field labels from display text into right justified, protected fields, you can assure that they will always be right next to their associated field. This is illustrated in Figure 16 below.

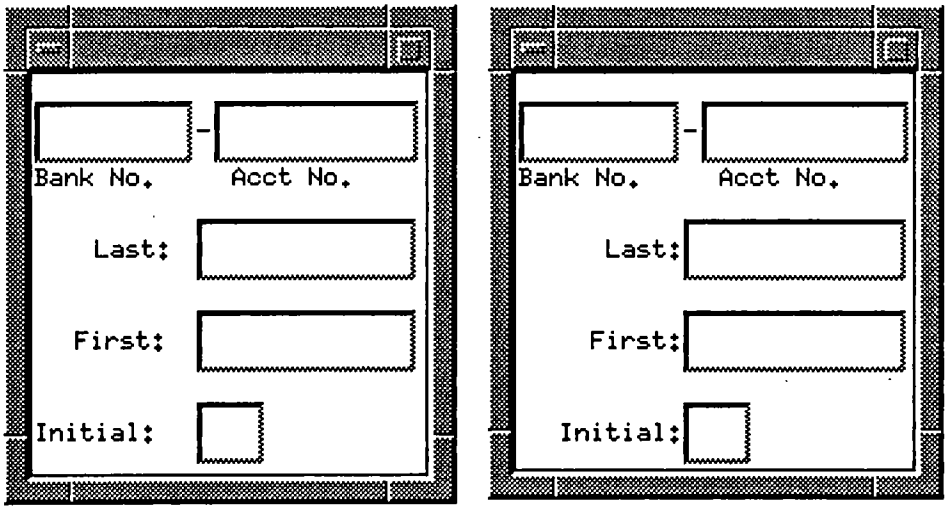


Figure 16: The screen on the left uses display text for the First, Last and Initial labels. The screen on the right uses right justified, protected fields.

The Bank No. field in Figure 16 stretches the grid. The first eight columns, which contain the field labels, stretch. In the screen on the left, the labels anchor in their starting cell, and consequently are no longer next to the fields that they correspond with. In addition, the colons at the end of each label don't line up. In the screen on the right, the labels have been converted into right justified, protected fields. They still look like display text, but they now anchor on the right in their ending cell, next to their corresponding fields.

3.3
WHITESPACE

If a widget does not horizontally fit in the cells it was drawn in, it expands into any unused cells (whitespace) around it before stretching the grid. Since a widget with an `halign` of `0` anchors on its left side, it can only expand into empty cells on its right. Similarly, a widget with an `halign` of `1` anchors on its right, and thus can only expand to its left.

Available whitespace is used up in proportion to `halign`. A widget with an `halign` of `.5` fills whitespace evenly on both sides. Expansion into whitespace based on `halign` assures that by default, left justified fields align on their left and right justified fields align on their right.

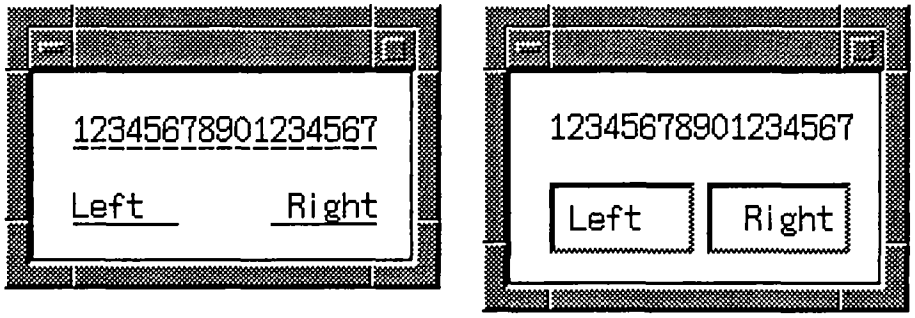


Figure 17: A screen with two fields of length six, shown in draw mode (left) and test mode (right). Left justified widgets expand into whitespace on their right. Right justified widgets expand into whitespace on their left.

Figure 17 illustrates how widgets appropriate whitespace. The screen contains two data entry fields of length six. The first field is left justified; it begins in column 1 and ends in column 6. The widget containing the field expands into the unoccupied space in columns 7 and 8. The second field is right justified. This field begins in column 12 and ends in column 17. Its widget expands leftward into columns 10 and 11.

The screen in Figure 18 below is the same as in Figure 17, except that there is a region of display text between the two data entry fields. Since there is no longer whitespace available, columns 1 – 6 and 12 – 17 stretch.

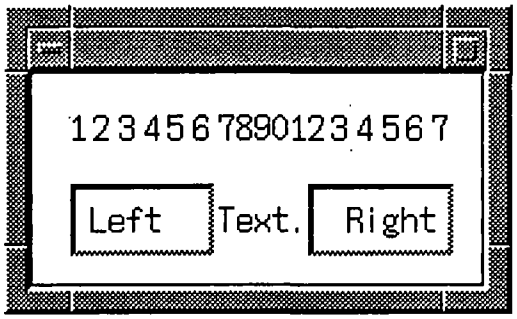


Figure 18: A screen with two fields of length six, and a region of display text. Since there is no room for the widgets to expand into, the grid stretches to accommodate them.

The numbers in individual protected fields at the top of the screen in Figure 18 indicate how the columns stretch. Notice that the extra space required for a widget is amortized evenly over the entire length of the widget.

3.4
PROPORTIONAL VS. FIXED WIDTH FONTS

The size of the grid in **JAM/Pi** is based on the average character width of the default screen font. There are two categories of fonts, *proportional* fonts and *fixed width* fonts.

```
In a fixed width font (like the Courier you are reading now) each
character occupies the same amount of horizontal space.
```

In a proportional font (like the Times Roman you are reading now) wider characters like "w", and capital letters occupy more space than narrow characters like "i" or "l".

In a fixed width font, the average character width is the width of each character. In a proportional font, the average character width is the mean width of all the characters in the font. The average character width of a proportional font is usually less than that of a comparably sized fixed width font, so the grid in a proportional font is smaller.

If a fixed width font is used throughout a screen, then text occupies the same amount of space as the cells available for it, provided that the grid has not stretched. This may be

desirable for applications converted from character **JAM**, since it tends to minimize the need to adjust alignment.

On the other hand, since proportional fonts take up less room than fixed width fonts, screen space can be economized without shrinking the font size by using a proportional font. Proportional fonts also enhance readability in large blocks of text.

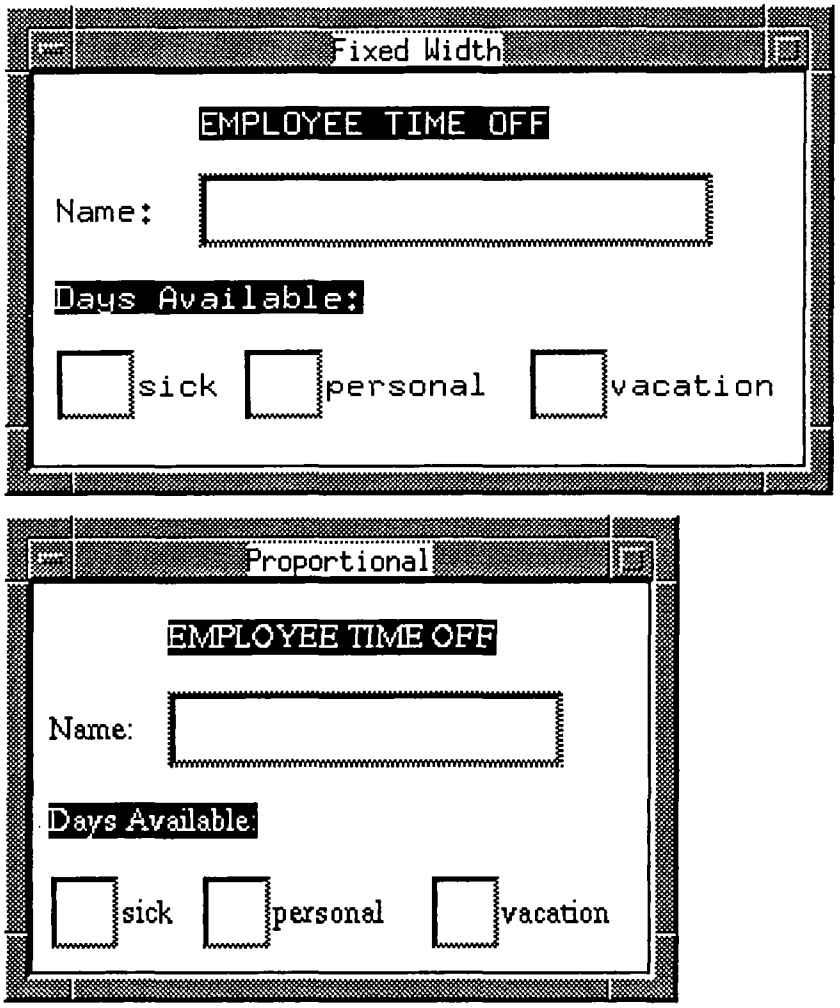


Figure 19: The same **JAM** screen in a 12 point fixed width font (top) and a 12 point proportional font (bottom).

The screens in Figure 19 demonstrate the size and alignment differences between proportional and fixed fonts. Notice that the proportional font makes for a smaller screen, but the spacing between items is inconsistent. For example, the horizontal white space between the first two fields at the bottom of the proportional screen is smaller than the white space between the second and third fields. These spaces can be adjusted with the `hoff` and `voff` field extensions (see section 3.6.3).

Screens may use a combination of proportional and fixed fonts. There is a default font for the application, and there may also be a default screen font and a font for an individual widget. Since the grid stretches but does not shrink, it is usually best to define the smallest font that you will use on a screen to be the default screen font. This strategy tends to make screens more compact by eliminating unnecessary whitespace.

3.5
WIDGET SIZE

The default size of a widget is based on the size of the field or region of display text, but is also influenced by other factors, including the font of the widget, and the border or other decorations around the widget. The font used in a widget is the default screen font, unless another font is specified as a field extension. The border and decorations around a widget depend upon the type of widget. The following sizing rule applies:

```
Width  =  (Avg_char_size_of_font  x  JAM_length) + Borders
Height =  Max_char_height_of_font + Borders
```

Since most widgets have a border, they are often wider than the grid cells allotted to them, and tend to stretch the grid horizontally unless there is at least one blank space available for them to expand into. Since vertical whitespace is not acquired by widgets, most widgets stretch the grid vertically as well.

If the text entered into a widget is wider than the widget, then the GUI shifts the text. For display–type widgets that cannot shift, if the above sizing rule does not leave enough room for the initial data, then the following rule is used instead:

```
Width  = Total_length_of_text + borders
Height = Max_char_height_of_font + borders
```

The default size of a widget may be overridden via the `height` and `width` field extensions. For details, refer to Chapters 5 and 6.

In P*i*/Motif and P*i*/OPEN LOOK, the size of the border and the type of decorations around a widget may be set in the resource file. Refer to Chapter 7.

3.6
FINE TUNING SCREEN ARRANGEMENT

Several additional field extensions are available for fine tuning the arrangement of JAM/Pi screens. These are `space`, for equally spacing array elements regardless of grid stretching; `noadj`, for turning off adjustment; and `hoff` and `voff`, for moving a widget horizontally and vertically.

3.6.1
The `space` Field Extension

Array elements are created as separate text widgets by default. These widgets are subject to the elastic grid. This means that there may be differences in the amount of space between the elements of an array, depending on how the grid has stretched. The `space` field extension guarantees that each element of an array has the same space between itself and the next element. The extension takes one argument, namely the space between each element.

For calculating its effect on the elastic grid, the total height of an equally spaced vertical array is the height of each element plus the space between elements. The row height of each element is then the total height of the array divided by the number of rows it occupies. The same is true for the total width and column width of a horizontal array. `space` is detailed in Chapters 5 and 6. An example screen is shown in Figure 20.

3.6.2
The `noadj` Field Extension

To override the elastic grid, use the `noadj` (called noadjust) field extension. Noadjust specifies that no grid stretching should be performed to account for a particular widget. Noadjust should be used with care, as it can cause widgets to overlap.

`noadj` takes a single string argument, either the word `rows` or the word `columns`. `noadj(rows)` turns off vertical grid stretching for the widget. `noadj(columns)` turns off horizontal grid stretching.

`noadj(rows)` is particularly useful to turn off vertical grid adjustment for very large widgets that have ample whitespace above or below them. It prevents a widget from upsetting the spacing between other objects on the screen and insures smooth screen scrolling for very large objects. `noadj(rows)` is often used in conjunction with `valign`, as shown in Figure 21.

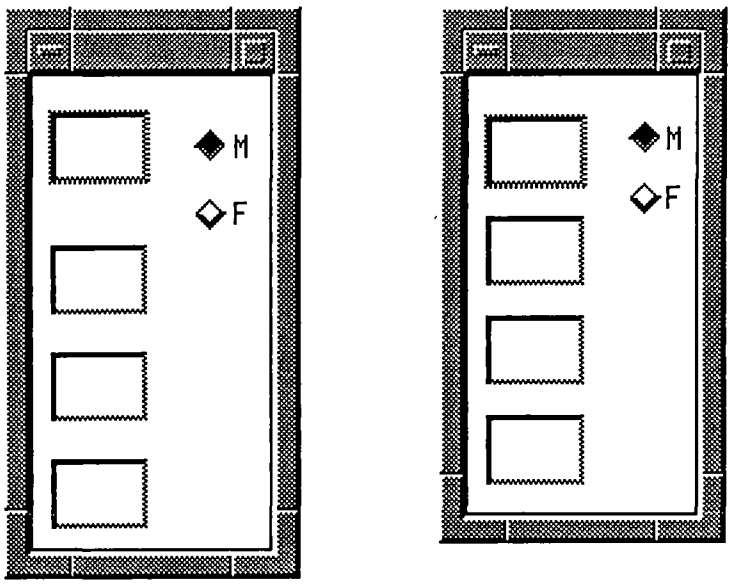


Figure 20: Two screens with a four element array and a radio button. The array is double spaced. The second group item in the radio button falls in the first blank row of the array. Its widget stretches this row. In the left hand screen, the result is an unequally spaced array. The array in the right hand screen has the `space` field extension, causing each element of the array to have the same space between itself and its neighbor. In this case, 10 pixels.

In the left screen of Figure 21, the BOOK push button stretches its row, causing uneven spacing between the Class, Rate and Avail. fields. In the right screen, BOOK has a vertical noadjust field extension that prevents it from stretching the grid. It also has a `valign` of `0`, anchoring it at the top, rather than at the center of its row. Without a `valign` of `0`, the push button would overlap the screen title bar.

Noadjust is less useful horizontally, since **JAM/Pi** uses up available horizontal white-space before stretching the grid. Since `noadj(columns)` disallows grid stretching for a widget, it almost always results in widgets overlapping.

3.6.3
The `hoff` and `voff` Field Extensions

To adjust a widget's position on the screen, use the `hoff` and `voff` (for horizontal and vertical offset) field extensions. `hoff` specifies the horizontal offset of a widget from its default placement. `voff` specifies the vertical offset. `hoff` and `voff` are applied

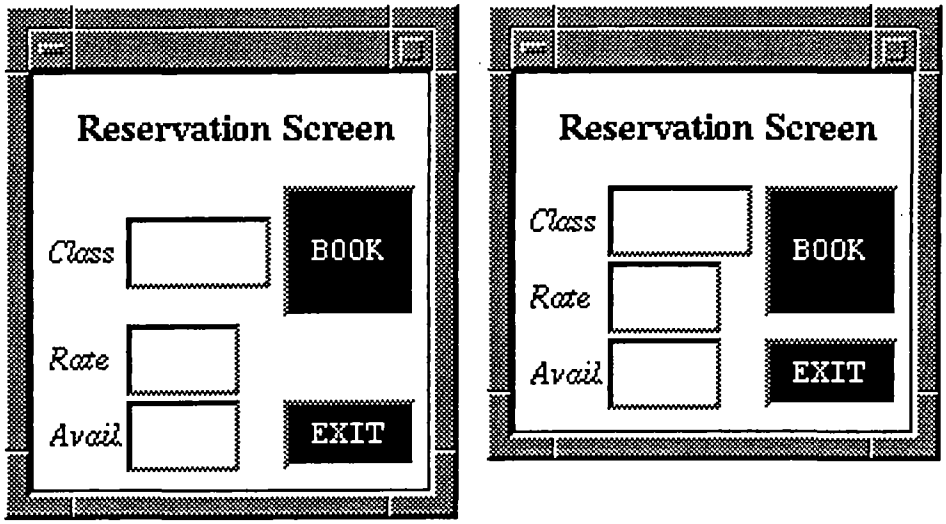


Figure 21: A screen where the `noadj(rows)` field extension is used to prevent a large button from stretching its row.

after any alignment or noadjust extensions. Therefore a widget with an `hoff` or `voff` still affects the grid as if it were in its default location, even though it is drawn elsewhere. These extensions should be used with care. They can cause widgets to overlap, and excessive use makes applications hard to maintain.

`hoff` and `voff` take a single argument, namely, a value indicating the amount to move. A signed value indicates movement relative to the widget's default position. An unsigned value indicates movement relative to the left side or top of the screen. The default unit of measurement is pixels. Alternatives such as inches, millimeters, characters, and grid units may also be specified.

For more information on `space`, `noadj`, `hoff`, and `voff`, refer to Chapters 5 and 6.

3.7
REFRESHING THE SCREEN

JAM calculates the positioning of objects only when a screen is first displayed. If a widget changes size or type while a screen is displayed, it may be necessary to recalculate the relative positioning of objects. This may be done via the `sm_adjust_area` library routine. For example, if the protections on a field change, a label widget can become a text widget. By not recalculating the screen, **JAM** avoids costly processing if the change is only temporary. Refer to Chapter 12 for details on `sm_adjust_area`

3.8
SEPARATOR ROWS AND COLUMNS

JAM/Pi provides screen extensions that create GUI lines and boxes to enhance screen appearance. Lines and box edges take up space, but the existence of a line or box should not affect the alignment of screen objects. Therefore, lines and boxes are not drawn within the regular grid cells. Instead, they are drawn in special separator rows and separator columns that appear between the rows and columns of the grid.

Separator rows and columns are created just wide enough to accommodate their contents, the edges of boxes and lines. Figure 22 illustrates how separator rows and columns relate to the elastic grid.

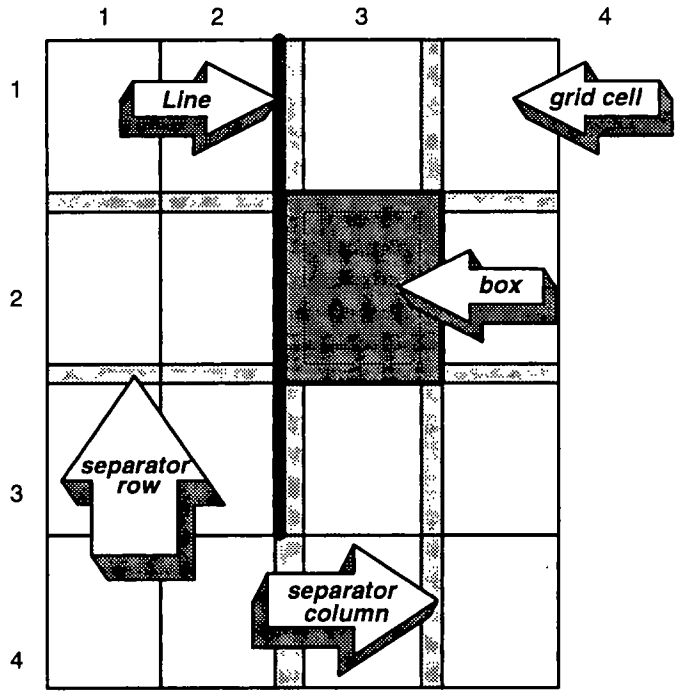


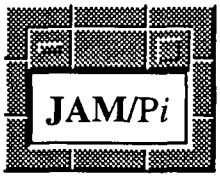
Figure 22: Screen containing two lines and a box. Lines and boxes are drawn in separator rows and columns that are just wide enough to contain the objects and their margins.

3.8.1

Separators and the Elastic Grid

The positioning algorithm considers lines and box edges to be non whitespace when calculating whether there is room for widgets to expand. Widgets can overlap lines or box edges, but only if they cross the row or column boundary containing the edge in draw mode. If the widget does not cross the boundary in draw mode, then the grid expands to prevent the widget from crossing the line or box edge. This strategy insures, for example, that a box intended to surround a set of fields surrounds those fields regardless of how large the widgets containing the fields become.

For information on how to create lines and boxes refer to Chapters 5 and 6.



Chapter 4
JAM Behavior in a GUI Environment

This chapter examines how the user interface in **JAM/P*i*** behaves, and describes some of the screen level features available in **JAM/P*i***.

4.1 JAM SCREENS

JAM screens each come up in their own GUI window. By default, the GUI window has a border and is fully decorated with resize and move handles, a minimize and maximize button, and a GUI window menu button. Scroll bars appear in the border only if they are necessary—ie., when the GUI window is too small to contain the **JAM** screen.

Screen extensions can be used to control various aspects of screen appearance and behavior. These include suppressing certain border decorations, starting the window as an icon, and specifying the title bar text.

4.1.1 Title Bars

The title bar on each screen contains the name of the file that the screen binary is stored in by default. For a title other than the file name, use the `title` screen extension. You may also suppress the title bar altogether with the `notitle` screen extension. See Chapters 5 and 6 for more on screen extensions.

M O In P*i*/Motif and P*i*/OPEN LOOK, title bar text may also be set through the resource file. For example, to change the title bar for a form called `mngform` in Motif, specify the following: `XJam*mngform.title: Title`

4.1.2

Multiple Document Interface in MS Windows

W*i*/Windows uses the Multiple Document Interface (MDI). This interface enables each **JAM** screen to be a fully functional Windows screen, and improves the coordination of **JAM** screens. The MDI specification places certain constraints on the user interface and screen structure for a Windows application. Other examples of MDI applications are the Program Manager and File Manager under Windows 3.

Under the MDI, an application is contained within a frame, or main screen. The space within the frame is used to display other screens within the application. These child screens are just like other Windows screens, except that they have no menu bar, and they are constrained from moving outside of the frame.

The following rules apply to screens within an MDI frame:

- Only one child screen at a time holds the focus.
- The menu bar across the top of the frame refers to the active screen and to the application as a whole.
- When an MDI screen is iconified, the icon appears at the bottom of the frame.
- When an MDI screen is maximized, the screen takes up the entire frame. The screen's title bar disappears, and the name of the screen is appended to the name of the application in the frame's title bar, as in:

  ```
  JAM - [mainscrn]
  ```

- The menu bar may have an additional item called "Window." This menu option allows the user to select and rearrange the various screens in the frame. Of course only screens that are siblings of the screen at the top of the **JAM** window stack may be made active.
- The title bar on the active screen is highlighted.

For more information on the Multiple Document Interface, see *Programming Windows: The Microsoft Guide to Writing Applications for Windows 3* by Charles Petzold, published by Microsoft Press; or the *Microsoft Windows Software Development Kit Guide to Programming*, published by Microsoft Corporation, which is distributed as part of the Software Development Kit.

Figure 23 shows **JAM**/P*i* under the Windows MDI.

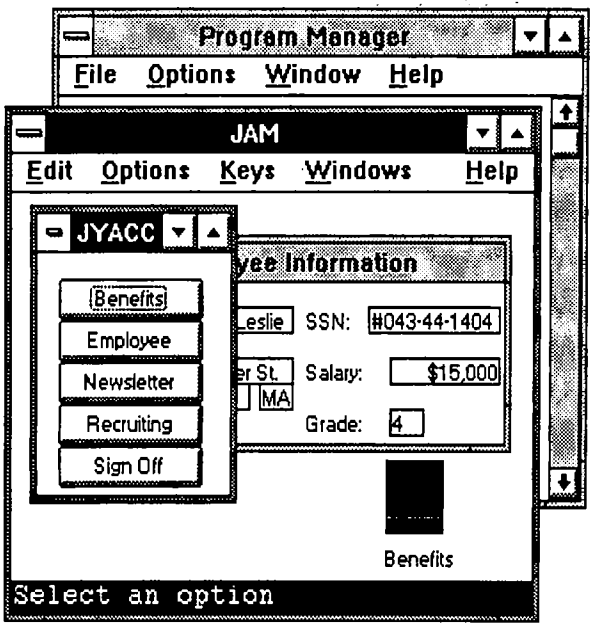


Figure 23: P*i*/Windows runs in an MDI frame with a single menu bar at the top and a single status line at the bottom. **JAM** screens are constrained to move only within the frame.

M **O** In P*i*/Motif and P*i*/OPEN LOOK, **JAM** screens are not constrained within a frame. There is however an optional base window that may be used to contain an application-wide menu bar and status line. Refer to Chapter 7.

4.1.3

Focus

Just as in character **JAM**, control flow is specified by the developer, using any combination of forms, windows and sibling windows. Although several screens may appear on the display at any given time, only the screen at the top of the window stack or one of its siblings may be made active.

A user may select a sibling window with a mouse click, or choose it by name from the optional `Window` heading on the menu bar. The names of all open screens appear under this heading, but only those that are siblings of the active screen may be selected.

An option in the resource or initialization file greys out text on inactive screens. Refer to Chapter 7.

Certain aspects of focus behavior are dictated by the GUI. These are detailed below.

W In Pi/Windows, when a screen is made active, it rises to the top and its title bar becomes highlighted. If the user attempts to activate a **JAM** screen that is neither a sibling nor at the top of the window stack, the screen rises to the the top when the mouse button is depressed, but then sinks back down when the button is released, and the former active screen retains the focus. This functionality allows dormant screens to be moved, resized and viewed, even though they cannot accept the focus.

M **O** In Pi/Motif and Pi/OPEN LOOK, the screen at the top of the **JAM** window stack has the keyboard focus in **JAM**. In order to best use **JAM**, we suggest that you activate the `XJam*focusAutoRaise` or `OLJam*focusAutoRaise` resource. This insures that when **JAM** has the keyboard focus, the active **JAM** screen appears on top of any other GUI windows on the display. The following entry sets this resource in Motif:

```
XJam*focusAutoRaise:        true
```

In OPEN LOOK it should be:

```
OLJam*focusAutoRaise:       true
```

NOTE: these resources are not the same as the `Mwm*focusAutoRaise` or `OLwm*focusAutoRaise` resources.

Motif and OPEN LOOK support two focus models, pointer focus and explicit focus. See your Motif or OPEN LOOK manual for details on specifying a focus model.

4.1.4

JAM Borders

JAM borders, specified in the Screen Attributes window, are ignored in **JAM**/Pi since the interface provides a border for each GUI window. The appearance of the GUI window border is controlled by the screen extensions.

4.1.5

Iconification

As a general rule, if you wish the user to iconify screens in your application, use sibling windows. The specifics of when the user may iconify screens are GUI dependent:

In Pi/Windows, active screens may be iconified if there is a sibling window available to accept the focus, or if the active screen is the **JAM** form. Iconifying stacked windows is not permitted, since the focus would be restricted to the iconified window, but some other window would be visible. This might confuse the user.

The `icon` screen extension associates an icon with a **JAM** screen. This icon must be listed in the MS Windows resource file that is compiled with your executable. **JAM** icons appear at the bottom of the MDI frame. When a window is iconified, the next sibling receives the focus. If no sibling windows are open, then the iconified window retains the focus.

The `iconify` screen extension, specifies that a screen should be started as an icon.

In Pi/Motif, individual windows may be iconified only if they have the `icon` screen extension. This extension associates a particular icon bitmap with the screen. Screens with this extension have a minimize button in the screen border and a minimize choice on the GUI window menu that is accessed via the menu button in the upper left hand corner of the screen border.

There are several resources available in Motif to manage icons. `Mwm*useIconBox` creates an icon box where application icons are stored. The `iconAutoPlace` and `iconPlacement` resources control the placement of icons when there is no icon box. Refer to the *OSF/Motif Programmer's Guide* for more information.

In Pi/OPEN LOOK, any window that has a window header may be iconified. Only transient windows, such as message windows cannot be individually iconified.

Preventing Iconification

The `nominimize` screen extension removes the minimize button and the minimize entry from the GUI window menu.

4.1.6

Toggling Between Menu Mode and Data Entry Mode

JAM/Pi allows the user to switch between menu mode and data entry mode on mixed use screens simply by clicking the mouse. Clicking on a push button toggles **JAM** into menu mode before processing the selection. Clicking on a text widget toggles **JAM** into data entry mode. This makes it very convenient to incorporate push buttons into your data entry screens. This behavior has been incorporated into character **JAM**.

4.2

ERROR AND STATUS MESSAGES

In **JAM/Pi**, status messages appear on the status line and messages requiring acknowledgement appear in dialog boxes. A dialog box is an application modal window: a user must deal with it before doing anything else in the application. The table below indicates where each type of message appears. Figure 24 illustrates the various dialogs.

Mode in JPL	*Equivalent C Function*	*Message Location*
`setbkstat`	`sm_setbkstat`	status line
`d_msg`	`sm_d_msg_line`	status line
`emsg`	`sm_emsg`	dialog box
`err_reset`	`sm_err_reset`	dialog box
`qui_msg`	`sm_qui_msg`	dialog box
`quiet`	`sm_quiet_err`	dialog box
`query`	`sm_query_msg`	"OK / Cancel" or "Yes/No" dialog box.

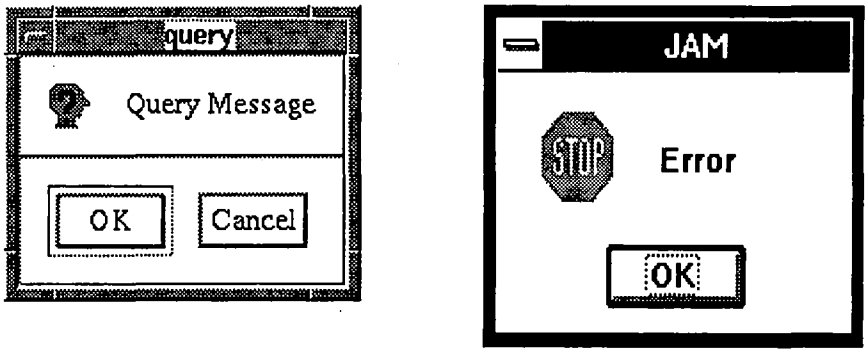


Figure 24: A Motif OK/Cancel dialog (left) and a Windows OK dialog (right).

To acknowledge an OK dialog box, click on the OK button or press the space bar (or other ER_ACK_KEY as specified in the setup file). In an "OK / Cancel" dialog box, click on either button or press SM_YES or SM_NO. The OK button returns SM_YES and the Cancel button returns SM_NO.

W In *Pi*/Windows a yes/no dialog box is used for query messages. The user must press y or n, not SM_YES or SM_NO.

4.2.1

Dialog Box Icons

A dialog box may have one of several icons on it. Specify the icon by prefacing the message with `%T`. The character immediately following the `%T` specifies the icon. The table below illustrates the icons.

Character	*Meaning*	*Motif Icon*	*Windows Icon*
`e`	Error	 Error	 **Error**
`i`	Information	 Information	 **Information**
`t`	Wait	 Wait	– Not available –
`w`	Warning	 Warning	 **Warning**

If there is no `%T` in the message string, then no icon appears. In OK/Cancel or Yes/No dialogs, a question mark icon appears by default. **JAM/Pi** cannot change this icon.

W In Pi/Windows, `qui_msg` and `quiet` message dialogs contain a stop sign by default if there is no `%T` in the message text.

M In Pi/Motif, you may specify `%T(`***iconname***`)`, where ***iconname*** is the name of an icon bitmap or pixmap. See the the man page for `XmGetPixmap` in the *OSF/Motif Programmer's Reference* for a listing of the path searched for bitmaps.

O In Pi/OPEN LOOK, dialog boxes do not support icons. `%T` strings in messages are ignored.

4.2.2
Location of the Status Line

W In P*i*/Windows, the status line appears at the bottom of the MDI frame. There is one status line per application. Individual screens do not have their own status line.

M **O** In P*i*/Motif and P*i*/OPEN LOOK, by default the status line appears in the same window that contains the menu bar. This is known as the "main" or "base" window.

The `formStatus` resource controls whether status messages appear only in the base window, or also in individual **JAM** screens. If `formStatus` is `false`, all status messages appear only in the base window. If `formStatus` is `true`, only background status messages appear in the base window. All other status messages (d_msg_line, wait, field and ready) appear at the bottom of the active **JAM** screen. The status line on inactive screens remains as it was when the screen was last active.

Note that the appearance of the base window is controlled by the `baseWindow` resource. This is documented in Chapter 7. If there is no base window, and `formStatus` is `true` then background status messages will be lost; if `formStatus` is `false`, then all status messages will be lost.

Status Line Keytops

Status line keytops work as they do in character **JAM**. For a more GUI compliant navigation tool, you may wish to use menu bars instead of keytops. See Chapter 8.

Keytop Functions in the Authoring Tool

Functions that appear on the status line in the authoring tool in character **JAM** appear in the menu bar or as keysets (depending upon which is enabled) in **JAM/***Pi*.

4.3
SHIFTING AND SCROLLING

JAM's user interface exhibits certain shifting and scrolling behavior. In addition, GUIs have their own shifting and scrolling behavior. This section explains how **JAM/***Pi* reconciles both these behaviors.

4.3.1

Shifting Fields and Proportional Fonts

In **JAM**/P*i*, the distinction between shifting and non–shifting fields becomes clouded, particularly when proportional fonts are used.

In character **JAM**, a field that has a maximum shifting length that is greater than its on–screen length is defined to be a shifting field. When the number of on–screen characters is reached, the field shifts to accommodate additional data, up to the shifting length.

In **JAM**/P*i*, the length of the actual data determines whether a widget shifts. Since the length of a text widget is determined by the average character size of the font, it is possible that a non–shifting field (in the **JAM** sense) may actually shift, if it happens to contain wide characters in a proportional font. It is also possible that a shifting field does not shift, even though it is full, because it happens to contain narrow characters.

These two cases are illustrated in Figure 25. Widget 1 is a "non–shifting" field of length ten. It shifts to accommodate the ten "W"s inside it. Widget 2 is a "shifting" field of length ten with a maximum shifting length of fifteen. It contains fifteen "i"s, but still has space left over, and therefore does not need to shift. Widget 3 is a field of length ten. Because it uses a fixed width font, it is sized to contain exactly ten characters regardless of which characters they are.

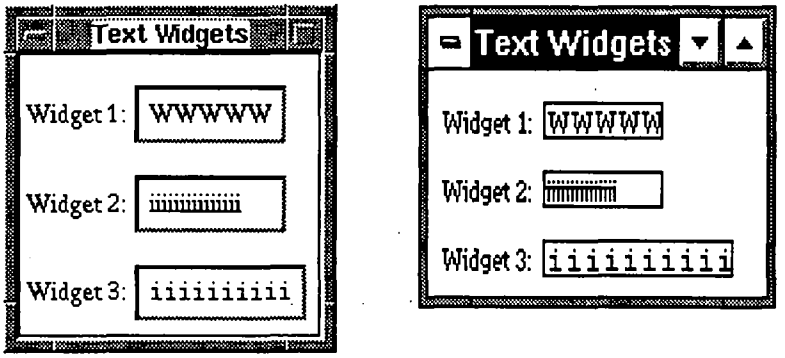


Figure 25: Text widgets in P*i*/Motif and P*i*/Windows.

4.3.2

User Interface to Shifting and Scrolling

A field may be shifted and scrolled in several ways. With the mouse, a user may shift or scroll a field by dragging the mouse cursor beyond the edge of the widget in the desired direction. If shifting or scrolling indicators are active, the user may click on these to shift or scroll a field. The following **JAM** logical keys shift fields via the keyboard: LSHF, RSHF, LARR and RARR. The following **JAM** logical keys scroll fields via the keyboard: SPGU, SPGD, UARR, DARR and NL.

Shifting or scrolling fields in multiline text widgets or list boxes may be shifted or scrolled via optional scroll bars.

4.3.3

Shifting and Scrolling Indicators

JAM scrolling indicators appear whenever an array may be scrolled. **JAM** shifting indicators appear only when a field requires shifting from **JAM**'s perspective—ie., when there are more characters in the field than the field's on–screen length.

Turning Off JAM Shift/Scroll Indicators

In **JAM/P*i***, you may wish to turn the **JAM** shifting and scrolling indicators off, as they don't conform to GUI style guides and may confuse end users. Use the `IND_OPTIONS` keyword in the Setup File to select the level of shift/scroll indication that you wish. There are four possible settings for this keyword, as described below:

- `IND_NONE` No indicators
- `IND_SHIFT` Shift indicators only
- `IND_SCROLL` Scroll indicators only
- `IND_BOTH` Shift and scroll indicators

The setup file is fully documented in the **JAM** *Configuration Guide*. Note that the value of `IND_OPTIONS` may also be changed at runtime, via the `sm_option` library routine.

We strongly suggest that you turn off shifting and scrolling indicators in P*i*/Windows, as they can cause alignment problems beyond being unsightly.

M In addition to the **JAM** shifting and scrolling indicators, Motif provides its own indicators that appear as arrow button widgets. These indicators may be turned off via the command line option `+ind`, or by setting `indicators` equal to `False` in the resource file. Refer to section 7.7.1 for more information.

At least one set of indicators (**JAM** or Motif) should be disabled in order to not confuse the end-user.

O In addition to the **JAM** shifting and scrolling indicators, OPEN LOOK provides its own indicators that appear as arrow buttons. These indicators cannot be turned off, so it is recommended that the **JAM** indicators be disabled, in order not to confuse the user.

Changing the Characters Used as Indicators

If you choose to use **JAM** shifting and scrolling indicators, you may wish change the characters that represent them. Depending on the character set of the font you are using, the default values may or may not appear to your liking. To change the shift/scroll indicator characters, you must alter the Video File. The ARROWS keyword controls these characters. Refer to the Video File chapter of the **JAM** *Configuration Guide* for details.

4.4 CUTTING, COPYING & PASTING TEXT

Within a text widget, the user may take advantage of the text cut, copy and paste features offered by the GUI. These features provide access to the clipboard maintained by the GUI, allowing inter–application text manipulation. For example, you can copy text from a **JAM** application and paste it into a word processor that also supports the GUI clipboard. Only text in text widgets may be manipulated in this way.

W In Pi/Windows, to select a range of text, drag across the field with the left mouse button depressed or use `Select All`. Choose `Cut` or `Copy` from the `Edit` heading on the Windows menu bar. Move the cursor to the desired location and choose `Paste` from the `Edit` heading on the menu bar. You may also use the keyboard shortcuts listed under the `Edit` heading.

NOTE: Pi/Windows allows text in the GUI clipboard to be *pasted* as display text in Draw Mode. It does not allow display text to be cut or copied, though.

M In *Pi*/Motif, to select a range of text, drag across the field with the left mouse button depressed. The selected text is highlighted, and becomes the "primary selection". Release the button and reposition the cursor. Click the left button to position the cursor at the desired new location, and then click the middle mouse button to paste the text at this cursor position.

Alternatively, if menu bars are enabled, you may use the `Edit` menu heading to select, delete, cut, copy and paste text. Note that the copy option on a **JAM** menu bar copies any highlighted text on the desktop, regardless of what application owns it. This allows for inter-application copying. The `Select All` option selects all the text in a single or multiline text widget. For more information on menu bars, see Chapter 8.

O In *Pi*/OPEN LOOK, to select a range of text, either drag across the field with the select mouse button depressed, or click the select mouse button at the start of the text and the adjust mouse button at the end of the text. The text must then be copied or cut before it can be pasted. This may be done either with the cut/copy keys, or the cut/copy selection on the pop-up edit menu. To paste the buffered text, click the select mouse button at the desired location and use the paste key or paste menu choice.

When pasting text into a widget, **JAM** enforces the field's character edits. **JAM** does not overflow the text into the next field if there is more text in the paste buffer than fits in the designated field. Overflow text is truncated.

When an area of text is selected, typing from the keyboard deletes the selected text. The first character typed replaces the highlighted text; subsequent characters are inserted in or overwrite the line, depending on whether you are in insert or overstrike mode.

Text that is not in a text widget *cannot* be edited via the GUI-provided cut and paste, although it can be manipulated via the **JAM** select mode feature in the screen editor. Select mode includes a clipboard for convenient cutting and pasting.

4.5 SOFT KEYS

Soft keys work as they do in character **JAM**. Soft key labels are converted into button widgets which can be clicked on with the mouse. Just as in character **JAM**, you must make the appropriate entries in the `main` routine (`jmain` or `jxmain`) and in the video file to activate soft keys. Refer to the **JAM** *Author's Guide* or *Configuration Guide* for more information. Soft keys should *not* be implemented using the "simulated" keyword

in the video file. This keyword is reserved for machines that don't provide support for either soft keys or push buttons.

W **NOTE:** Soft keys are not currently implemented in Pi/Windows.

4.5.1

Location of Soft Keys

M O Soft keys appear by default on the base screen. The `formMenus` resource determines whether they also appear on individual screens. If this resource is set to `false`, the default, then individual screens do not have their own keysets. If it is `true`, then keysets with a scope of `KS_FORM` and `KS_OVERRIDE` appear on the current screen, while those with a scope of `KS_SYSTEM` and `KS_APPLIC` appear on the base screen.

The appearance of the base screen is controlled by the `baseWindow` resource. If there is no base screen, then any keysets that would normally appear there are lost.

4.5.2

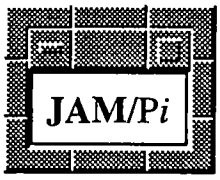
Soft Keys vs. Menu Bars

Soft keys and menu bars are mutually exclusive, because they share the same programmatic hooks. The developer must choose whether to use one or the other. The selection of soft keys versus menu bars is made in the `main` routine, either `jmain.c` or `jxmain.c`, by initializing either soft key support or menu bar support. If an application is to use keysets in character **JAM** and menu bars in **JAM/Pi**, then the `main` routine should call the soft key initialization routine before it calls the menu bar initialization routine. The second library call will override the first in **JAM/Pi**, but will be ignored in character **JAM**.

If you are using menu bars on some platforms and keysets on others, you may wish to use libraries to store the keyset and menu bar files. This strategy is explained in section 8.9.

The `kset2mnu` Utility

The `kset2mnu` utility converts keysets into menu bars. This is useful for porting character **JAM** applications developed with soft keys into **JAM/Pi** applications that use menu bars. For an explanation of how to implement menu bars and convert keysets into menus, refer to section 8.9. For a description of the `kset2mnu` utility, see section 12.2.



Chapter 5
Entering Screen and Field Extensions

Field and screen extensions provide access to the multitude of features available under GUI's. Here the developer may specify fonts, colors, window decorations, positioning, and specialized widgets. This chapter discusses how to enter screen and field extensions into the formatted screens provided by **JAM/P*i***. Chapter 6 is a reference for the extensions.

5.1 INTRODUCTION

Screen and field extensions are stored in the JPL module comments associated with screens and fields. Extensions may be entered directly into the JPL module, or they may be entered into special screens provided with **JAM/P*i***. Entering extensions into the formatted screens is more convenient than entering them directly into the JPL comments.

- The SPF11 key opens the screen extensions window. The scope of a screen extension is the current screen.
- The SPF12 key opens the field extensions window. The scope of a field extension is the current field.

When either of these screens is opened, the extensions stored in the JPL comments are read, and the screen is filled in with any relevant data. When the screen is closed with the transmit key or OK button, changes to the extensions are written back into the JPL comments.

For field extensions, any changes made to a widget type that are inconsistent with the edits on the underlying **JAM** field cause the **JAM** field edits to be updated when the extensions screen closes.

This chapter describes the formatted screens, and briefly discusses each extension. Chapter 6 is a reference chapter for screen and field extensions, with a man page for each extension. Refer to Chapter 6 for any details not covered in this chapter.

The values entered as arguments to the various extensions may be colon expanded variables. This is discussed in section 6.2.1.

NOTE: The name of each extension as it appears in the JPL is noted alongside each entry in this chapter. This way it may be easily referenced in Chapter 6.

5.2
THE SCREEN EXTENSIONS WINDOW

To open the screen extensions window, press SPF11. The window that appears is shown in Figure 26. The following options are available:

- title? (`title`)

  Select yes or no. If you select no, the screen name (with the extension stripped off) is used as the title. If you select yes, a data entry field appears for you to fill in with the title text. For a blank title, leave this data entry field blank.

- icon (`icon`)

  Enter the name of the icon to use when this screen is minimized. Specify the full path if the icon is not on the icon search path used by the GUI. If no entry is made, then the screen cannot be iconified. If the specified icon is not found, the default icon is used.

- font (`font`)

  Enter the name of the default screen font. This font is used for display text and widgets that don't have a font of their own. The font name may be either a GUI font specification or a GUI independent font alias. Press the **JAM** HELP key, or choose Help from the menu bar to bring up an item selection screen containing a list of font aliases defined in the resource file. Select a font alias from this list or choose "custom fonts" to bring up a font selection screen to search for a GUI dependent font. See Figure 27.

- foreground (`fg`)

  Specify the default foreground color for this screen. The default foreground color overrides any unhighlighted white foregrounds on the screen. Enter the name of a GUI color or a GUI independent alias. Press the **JAM** HELP key, or choose Help from the menu bar to bring up an item selection screen containing a list of color aliases defined in the resource file. Select a color alias from this list or choose "custom colors" to bring up a color selection screen to search for a GUI dependent color. See Figure 28.

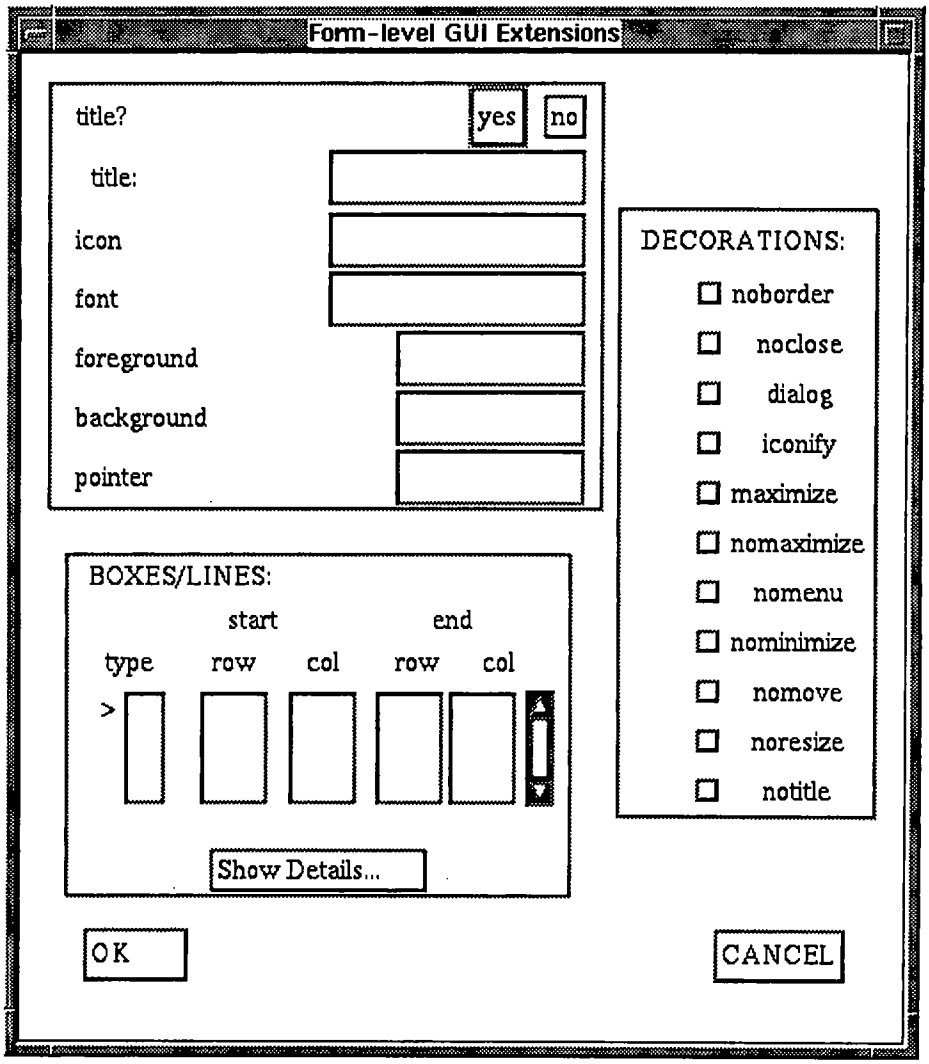


Figure 26: The Screen Extensions window.

- background (bg)

  Specify the default background color for this screen. The default background color overrides the screen's background color, and any background on the screen whose display attributes match the screen background. Enter the name of a color, or press the HELP key for a list of aliases.

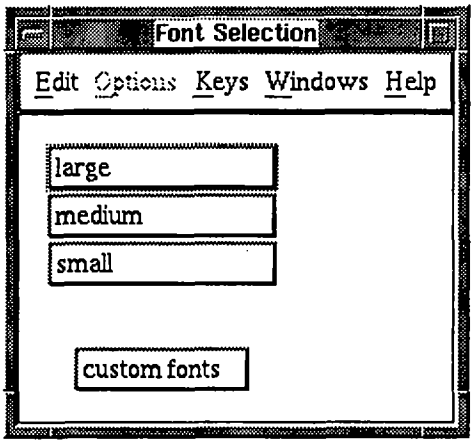


Figure 27: An item selection screen with a list of user–defined font aliases.

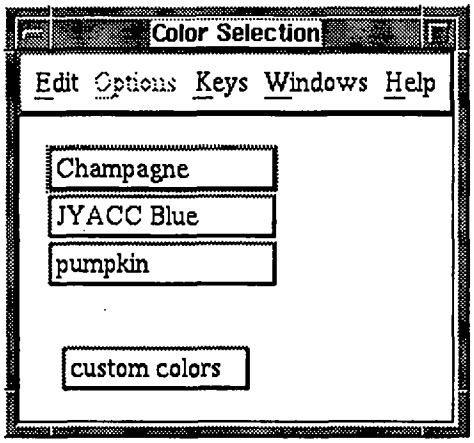


Figure 28: An item selection screen with a list of user–defined color aliases.

- pointer (`pointer`)
  Enter the name of the pointer shape to use on this screen. The default pointer is an arrow.

- Decorations
  The following options may be set regarding the decorations on the GUI window border:

- noborder (`noborder`)
  Eliminate the GUI border, removing the resize handles, title bar, and maximize and minimize buttons, leaving only a thin bounding box.
- noclose (`noclose`)
  Suppress the close option on the GUI window menu.
- dialog (`dialog`)
  Make this screen into a dialog box. A dialog box is an application modal window that cannot be resized, maximized or minimized. This is not supported in P*i*/Motif.
- iconify (`iconify`)
  Start screen as an icon.
- maximize (`maximize`)
  Start screen maximized.
- nomaximize (`nomaximize`)
  Prevent screen from being maximized by removing the maximize button and the maximize option on the GUI window menu.
- nomenu (`nomenu`)
  Eliminate the GUI window menu.
- nominimize (`nominimize`)
  Prevent screen from being minimized by removing the minimize button and the minimize option on the GUI window menu.
- nomove (`nomove`)
  Suppress the move option on the GUI window menu. This option *does not* prevent the user from moving the window with the mouse.
- noresize (`noresize`)
  Prevent this screen from being resized by removing the resize handles and the size option on the GUI window menu.
- notitle (`notitle`)
  Eliminate the title bar, including the minimize, maximize and GUI window buttons. To eliminate only the title text, use `title()`.

- Boxes and Lines (`box`, `hline`, `vline`)
  Boxes and lines may be drawn on the screen by filling in the appropriate information in the fields described below:
  - type — Enter `B` for a box, `H` for a horizontal line, or `V` for a vertical line.
  - start row — Enter the starting row for the line or box.
  - start column — Enter the starting column for the line or box.

- end row — Enter the ending row for the line or box. If `type` is a horizontal line, then this field is protected from data entry.
- end column — Enter the ending column for the line or box. If `type` is a vertical line, then this field is protected from data entry.
- Show details — Click on this button to set the display details for the line or box. A different screen appears depending on which type of object is selected. The details window is described below.

5.2.1

The Details Window for Lines and Boxes

Specify the appearance of a line or box by popping up the details window, described below. A sample details window is shown in Figure 29. The items on this screen provide the arguments to the `hline`, `vline`, and `box` screen extensions.

- Row/Column
  The row and column fields are the same as the row and column fields on the main screen extensions window. On this screen, though, only those fields that are appropriate for the type of object appear.
- Style — Choose a style from the option menu. Styles are GUI dependent. If the specified style is not supported, the default style is used instead.

Pi/Windows line styles are listed below. The default style is `single`.

`single` `dash` `dot` `dashdot` `dashdotdot`

Pi/Windows box styles are listed below. The default style is `single`.

`single` `dash` `dot` `dashdot` `dashdotdot`

If a color is specified for a box in Pi/Windows, then the style is ignored.

Pi/Motif line styles are listed below. The default style is `single`.

`single` `dash` `double dash` `etched in` `etched out`

Pi/Motif box styles are listed below. The default style is `etched in`.

`etched in` `etched out` `in` `out`

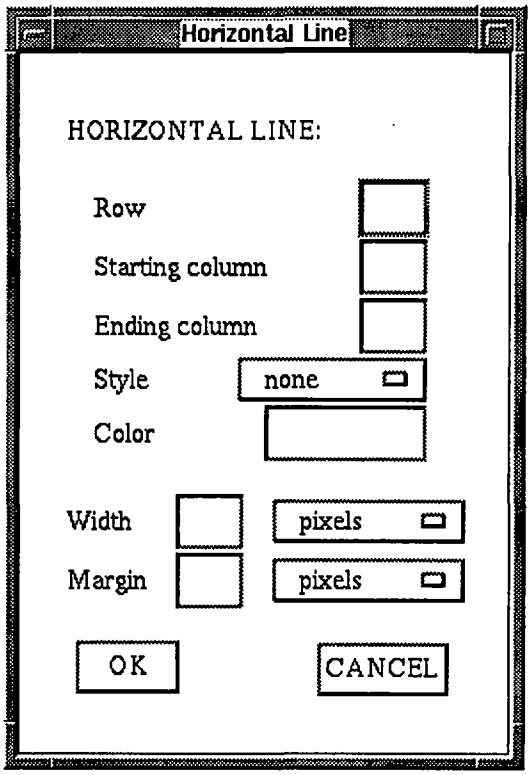


Figure 29: The details window for a horizontal line. There are similar windows for vertical lines and boxes.

O *Pi*/OPEN LOOK line styles are listed below. The default is `single`

```
single   dash
```

Pi/OPEN LOOK supports only a single line as the border for a box. The style is:

```
single
```

- Color — Enter a color for the line or box. Color may be a GUI color or a GUI independent color alias. Press HELP for a list of color aliases.

W In *Pi*/Windows, if you specify a color for a box, the style is ignored.

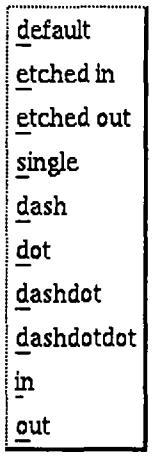


Figure 30: The styles option menu.

- Width — Enter the width of the line or the matte width of the box. For certain line styles the width is ignored. Refer to Chapter 6 for details.

  Choose the units for the value you've entered from the option menu to the field's right. The list is shown in Figure 31. Available units are:

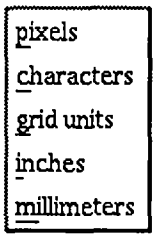


Figure 31: The units option menu.

  - pixels — The value is in screen pixels.
  - characters — The value is in character units. One character unit is the average character width of the default screen font.
  - grid units — The value is in grid units. Grid units are based on the average character width of the default screen font. For screen extensions, grid units and characters are the same.
  - inches — The value is in inches. In order to use inches, the X server must know the dimensions of your physical display.
  - millimeters — The value is in millimeters. In order to use millimeters, the X server must know the dimensions of your physical display.

- Margin This defines a blank margin around the outside of the line or box. Choose the units for the value you've entered from the option menu to the field's right.

5.3
THE FIELD EXTENSIONS WINDOW

The field extensions window allows you to set the details for a widget. Each type of **JAM** field has a default widget type associated with it. Use this screen to change the widget type of a field or set the font, colors, frame, size and alignment of a widget.

Each widget type has a Details screen associated with it, where you can set options specific to that widget, like scroll bars on a list box, or a pixmap on a push button. A sample field at the bottom of the extensions screen illustrates the extensions you've chosen.

5.3.1
Synchronizing JAM and the GUI

JAM/P*i* attempts to keep **JAM** synchronized and consistent with the GUI options you've chosen. If you change the widget type for a field, and that widget type is inconsistent with the **JAM** field edits, **JAM/P*i*** forces you to adjust the **JAM** field edits when you transmit out of the extensions screen. This prevents you from creating undesirable effects, like having a push button represent a field that is not a selection field.

If the option, "prompt for **JAM** field adjustments," is selected, **JAM/P*i*** asks you whether you want to adjust each relevant edit upon transmitting out of the screen. If this option is not selected, **JAM/P*i*** makes the adjustments without consulting you.

5.3.2
Forcing the Widget Type

If the option "force widget type" is selected, **JAM/P*i*** creates a field extension associating the widget type with the field, even if the widget type selected is the default widget type for that field. So, for example, an unprotected data entry field would get a `text` field extension, even though `text` is the default widget type for data entry fields. If this option is not selected, the widget type of the field can change depending on the **JAM** field edits, so subsequently protecting a data entry field would make it a label widget.

Be careful to satisfy **JAM**'s requirements for field behavior if you force a widget type. For example group items and menus must have text in them in order to be selectable.

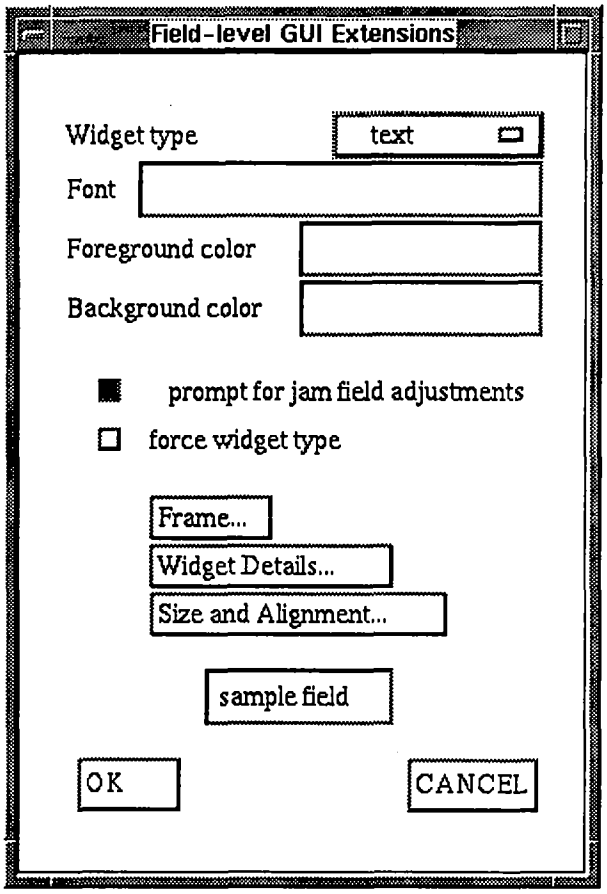


Figure 32: The Field Extensions window.

5.3.3

Entering Data in the Field Extensions Window

To open the field extensions window, move the cursor to a field and press SPF12. The window that appears is shown in Figure 32. The following options are available:

- Widget type

  Each type of **JAM** object has a default GUI widget that it transforms into. The default widget appears as the initial value in this field. Pop up the op-

tion menu to specify a widget other than the default. The list of widgets appears in Figure 33. Available widget types are:

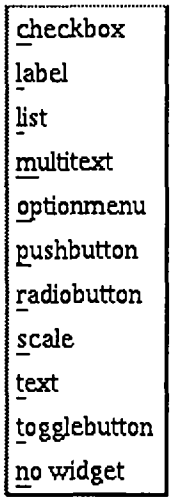


Figure 33: The widget type option menu.

- checkbox (`checkbox`)
  Create a checklist style toggle button widget from this field. This widget is the default for **JAM** checklist groups with boxes. This extension can be applied only to a group. A radio button group with this extension still acts like a radio button, it only appears as a checklist. Use the widget details window to replace the label text on the toggle button with a pixmap or to create a multiline label for the widget.
- label (`label`)
  Create a label widget from this field. Label widgets should be used for display text and protected fields. They do not support data entry or tabbing. Use the widget details window to replace the label with a pixmap or to create a multiline label.
- list (`list`)
  Create a list box widget from this field. List boxes are most appropriate for selection criteria like checklists, radio buttons, or menus on item selection screens. Use the widget details window to turn scroll bars on or off for the widget.
- multitext (`multitext`)
  Create a multiline text widget from this field. Multiline text widgets are most appropriate for arrays. The number of lines in the multiline

text widget is determined by the number of on–screen elements in the array. If the array is scrolling, the widget will scroll as well. Use the widget details window to turn scroll bars on or off for the widget.

- optionmenu (`optionmenu`)
  Create an option menu widget from this field. An option menu presents the user with a list of options from which to fill a field. The field should be either a cycle field (a scrolling array with one element) or a simple non–scrolling field. The off–screen occurrences of a cycle field can be used as the list of options. Alternatively, the list of options for the widget may be pulled from some other screen, much like an item selection screen. Set this behavior in the widget details window.

- pushbutton (`pushbutton`)
  Create a push button widget from this field. Push buttons are normally associated with protected menu fields since they are used as selection criteria. Use the widget details window to replace the label text on the push button with a pixmap or to create a multiline label.

- radiobutton (`radiobutton`)
  Create a radio style toggle button widget from this field. This widget is the default for **JAM** radio button groups with boxes. This extension can be applied only to a group. A checklist group with this extension still acts like a checklist, it only appears as a radio button. Use the widget details window to replace the label text on the toggle button with a pixmap or to create a multiline label for the widget.

- scale (`scale`)
  Create a scale widget from this field. Scales are appropriate for numeric fields whose contents are chosen from a range of values. Use the widget details window to input the range and number of decimal places.

- text (`text`)
  Create a text widget from this field. Text widgets are the default widget for unprotected fields. This extension allows you to turn a protected field into a text widget, but the widget's tabbing and data entry behavior is still dictated by the field's protections.

- togglebutton (`togglebutton`)
  Create a toggle button widget without checkboxes from this field. This widget is the default for **JAM** radio button or checklist groups without boxes. Use the widget details window to replace the label text on the toggle button with a pixmap or to create a multiline label.

- no widget (`nowidget`)

  Do not create a widget for this field. This is the default for fully protected non–display fields like menu control fields.

- Font (`font`)

  Specify the font name for the widget. If no font is specified, the default screen font is used. The font name may be either a GUI font specification or a GUI independent font alias. Press the **JAM** HELP key, or choose Help from the menu bar to bring up an item selection screen containing a list of font aliases defined in the resource file. From the item selection screen, choose an alias or choose "custom fonts" to bring up a file selection box to search for a GUI dependent font. See Figure 27 in the previous section.

- Foreground color/Background color (`fg`, `bg`)

  Specify the foreground and background colors for the widget. If no colors are specified, the default screen foreground and background colors are used. The colors may be either GUI color names or GUI independent color aliases. Press the **JAM** HELP key, or choose Help from the menu bar to bring up an item selection screen containing a list of color aliases defined in the resource file. From the item selection screen, choose an alias or choose "custom colors" to bring up a file selection box to search for a GUI dependent color. See Figure 28 in the previous section for an illustration.

- Prompt for JAM field adjustments

  This item is important only if you've changed the widget type of the field from its default value.

  If this toggle is set and there is an inconsistency between the **JAM** field edits and the widget type you've selected, **JAM/P*i*** prompts you with a dialog box asking whether you wish to alter the **JAM** edits on the field to match the widget type. The dialog box appears when you attempt to transmit out of the screen. Some inconsistencies may be ignored, while others must be changed. The buttons in the dialog box indicate whether a change is necessary or may be ignored. Figure 34 illustrates a sample field adjustment dialog box. If you choose not to make a required change, you are returned to the field extensions screen.

  If the "prompt..." toggle is not set, **JAM/P*i*** makes the changes to the **JAM** field edits upon transmitting out of the screen without consulting you.

- Force widget type

  This item is important when you have not changed the widget type from its default. If this toggle button is set, **JAM/P*i*** creates a field extension that forces this widget type on the field. If the protections or edits on the field subsequently change, the widget type does not change. If this option is not

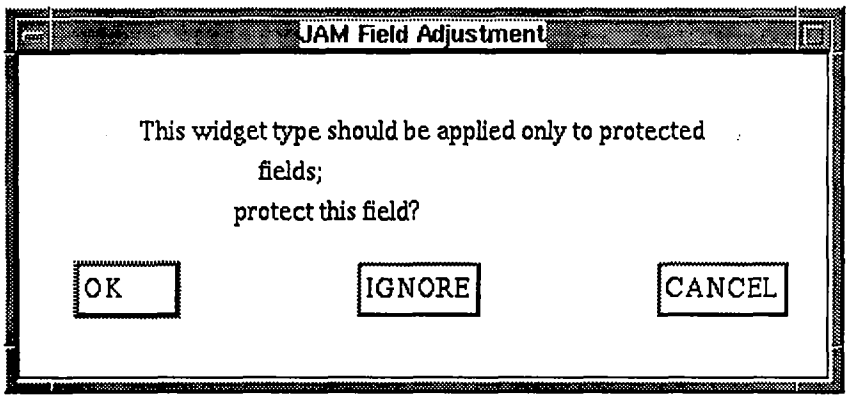


Figure 34: A sample field adjustment dialog box.

set, no extension is written to the JPL, and the field changes its widget type depending upon its edits.

If you have changed the widget type from its default, **JAM**/P*i* forces this option to be set.

5.3.4

The Frame Window

You can create a frame around a widget by pressing the frame push button to pop up the field frame specifications window shown in Figure 35. This creates a `frame` field extension. The following options set the arguments to the extension:

- Style — Choose a style from the option menu. Styles are GUI dependent. If the specified style is not supported, the default style is used instead. See Figure 30 in the previous section for an illustration.

W P*i*/Windows frame styles are listed below. The default style is `single`.

`single` `dash` `dot` `dashdot` `dashdotdot`

If a color is specified for a frame in P*i*/Windows, then the style is ignored.

M P*i*/Motif frame styles are listed below. The default style is `etched in`.

`etched in` `etched out` `in` `out`

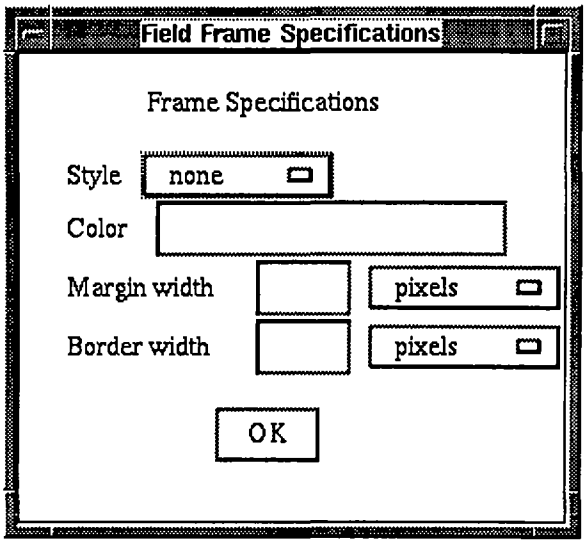


Figure 35: The field frame specifications window.

O Pi/OPEN LOOK frame styles are listed below. The default style is `single`.

`single`

- Color — Enter a color for the frame. Color may be a GUI color or a GUI independent color alias. Press HELP for a list of color aliases.

W In Pi/Windows, if you specify a color for the frame, the style is ignored.

- Margin — This is the width of a blank margin area around the outside of the frame. See Chapter 6 for details. Choose the units for the value you've entered from the option menu to the field's right. The list is shown in Figure 31 in the previous section. Available units are:
  - pixels — The value is in screen pixels.
  - characters — The value is in character units. One character unit is the average character width of the widget's font.
  - grid units — The value is in grid units. Grid units are based on the average character width of the default screen font.
  - inches — The value is in inches. In order to use inches, the X server must know the dimensions of your physical display.

- millimeters The value is in millimeters. In order to use millimeters, the X server must know the dimensions of your physical display.

- Border Enter the matte width of the frame. The matte is the area between the edge of the widget and the edge of the frame. Frames are drawn within the grid, so a frame with a wide matte or margin stretches the grid.

5.3.5

Widget Details Windows

Each widget type (except text) has an associated widget details screen with settings appropriate for the particular widget. The various screens are described below.

- Default Details screen

  The widget details screen for checklists, labels, push buttons, radio buttons and toggle buttons is illustrated in Figure 36.

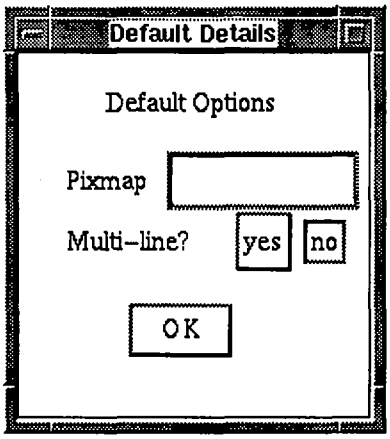


Figure 36: The widget details screen for: checklists, labels, push buttons, radio buttons and toggle buttons.

- pixmap (`pixmap`)

  Enter the name of a pixmap or bitmap file to display in the widget instead of the field's contents. See `pixmap` in Chapter 6 for details.

- multiline (`multiline`)

  Specify whether the widget should have multiple lines of text. The additional lines are held in the off-screen shifting length of the field. See `multiline` in Chapter 6 for details.

- List and Multitext Details screen

  These widgets can have scroll bars as an option. The level of scrolling is set in the arguments to the `list` or `multitext` extension. The details screen, shown in Figure 37, sets these arguments, controlling when scroll bars appear.

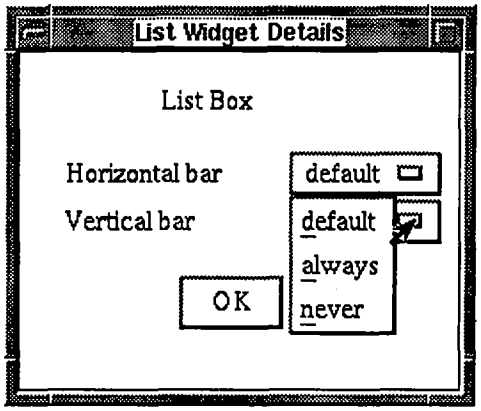


Figure 37: The widget details screen for list boxes and multiline text widgets. Notice that the option menu for vertical bar is posted.

  - Horizontal bar

    There are three options: `default`, `always`, and `never`:

    – `default` posts the scroll bar only when the field is a shifting field.

    – `always` posts the scroll bar regardless of need.

    – `never` posts no scroll bar.

  - Vertical bar

    There are three options: `default`, `always`, and `never`:

    – `default` posts the scroll bar only for a scrolling field.

    – `always` posts the scroll bar regardless of need.

    – `never` posts no scroll bar.

- Scale Widget Details screen

  Use the details screen to enter the arguments to the `scale` extension. These are the lower limit, upper limit, and number of decimal places in the scale's range. The screen is shown in Figure 38.

  - Lower limit Enter the lower bound of the range. The default is 0.
  - Upper limit Enter the upper bound of the range. The default is 100.

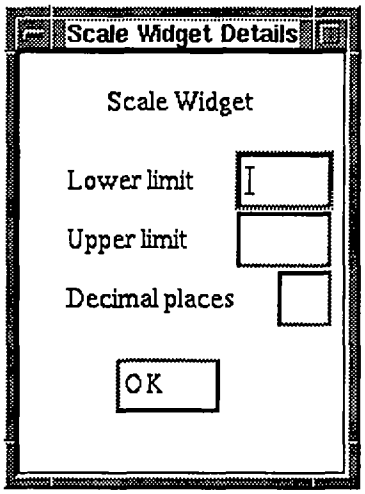


Figure 38: The details screen for a scale widget.

- Decimal places

  Enter the number of decimal places to use in the value. The default is 0 (whole numbers).

- Optionmenu Widget Details Screen

  Depending upon the arguments to the `optionmenu` extension, an optionmenu may be populated in one of two ways:

  With no arguments, an optionmenu is populated from the offscreen occurrences of the field. In this case the details screen is not needed. The field containing the optionmenu should be a scrolling array with one element

  If the field is not an array, the option menu is populated from menu fields on another screen, similar to an item selection screen. The arguments indicate the screen name and when the screen should be initialized. The optionmenu details screen sets these arguments. It is shown in Figure 39.

  - Form name — To populate the option menu from another screen, enter the screen's name here. Menu fields on the specified screen become items on the option menu.

  - Initialize? — The screen containing the options must be initialized before the option menu pops up. Initialization consists of opening and closing the screen and writing the values to the option menu widget. Initialization may be done at screen entry or each time the option menu pops up (or both).

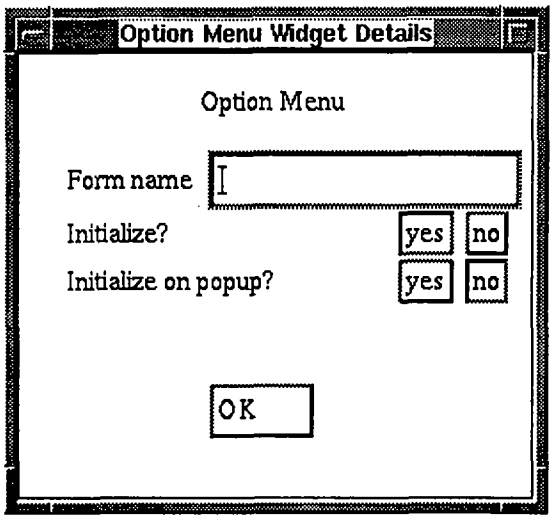


Figure 39: The option menu widget detail screen.

Set this to yes if you wish to initialize the optionmenu at screen entry.

- Initialize on popup?
  Set this to yes if you wish to initialize the option menu each time the option menu field is entered.

5.3.6

The Size and Alignment Window

JAM/P*i* gives each widget a default size, and places each widget on the screen in accordance with an algorithm based on the concept of an elastic grid. This algorithm is explained in detail in Chapter 3. The size and alignment window is for fine tuning the size and placement of widgets. Adjusting the placement of widgets is best done after all the widgets on a screen have been created and sized, since new widgets can affect the alignment of existing widgets. It is usually best to keep alignment settings to a minimum, as they can make a screen inflexible and hard to maintain. The size and alignment window is shown in Figure 40. The following options are available:

- height (`height`)
  Enter the height of the widget in this field and select the units for the height from the option menu to the field's right. Units are listed on page 67.

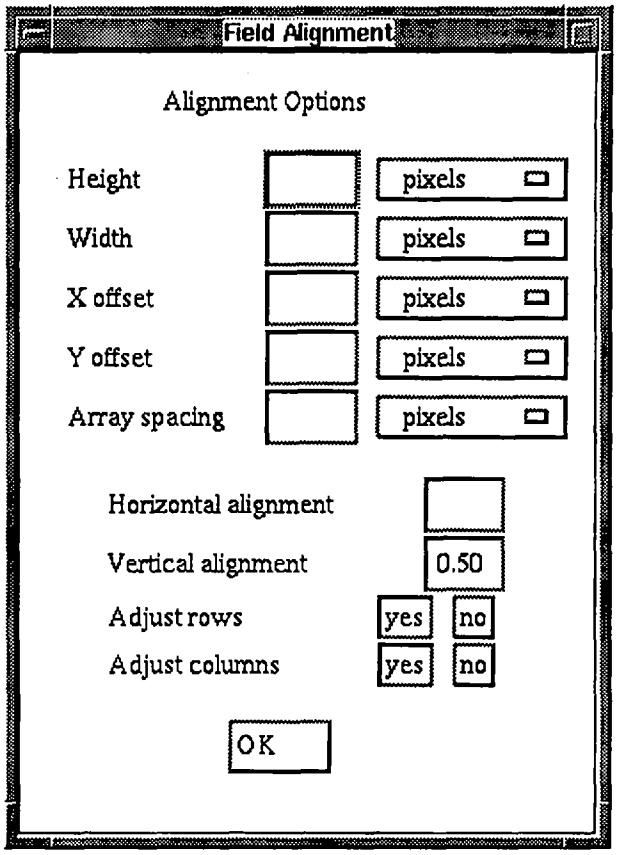


Figure 40: Screen for entering field size and alignment options.

- width (`width`)

  Enter the width of the widget in this field and select the units for the width from the option menu to the field's right. Units are listed on page 67.

- H offset (`hoff`)

  Enter the horizontal placement of the widget. An unsigned value indicates placement relative to the left margin. A signed value indicates a distance to move the widget relative to its default position. A positive signed value moves the widget the specified distance to the right of its default position, a negative value moves it to the left. The offset is calculated after the positioning algorithm has done its work, so this extension can cause widgets to overlap or run off the edge of the screen.

- V offset (`voff`)

  Enter the vertical placement of the widget. An unsigned value indicates placement relative to the top margin. A signed value indicates a distance to move the widget relative to it default position. A positive signed value moves the widget the specified distance down from its default position, a negative value moves it up. The offset is calculated after the positioning algorithm has done its work, so this extension can cause widgets to overlap or run off the edge of the screen.

- Array Spacing (`space`)

  Enter the amount of space to leave between array elements that appear as separate text widgets. Sometimes array elements are spaced unevenly due to grid stretching. Entering a value here assures that each element in the array is evenly spaced.

- Horizontal alignment (`halign`)

  Specify where this widget should anchor if it is narrower or wider than its grid cells. A widget will be narrower than its grid cells if another widget caused the grid to stretch horizontally. It will be wider than its grid cells if the option "Adjust columns" is set to no. See Chapter 3 for details.

  Enter a value between 0 and 1. 0 means that the left edge of the widget anchors in its starting cell (left alignment). 1 means that the right edge of the widget anchors in its ending cell (right alignment). Decimal values between 0 and 1 mean that the widget should align proportionally between its starting and ending cells. For example, .5 indicates center alignment. The default is 0 for left justified widgets, and 1 for right justified widgets.

- Vertical alignment (`valign`)

  Specify where this widget should anchor if it is shorter or taller than its grid cells. A widget will be shorter than its grid cells if another widget caused the grid to stretch vertically. It will be taller than its grid cells if the option "Adjust rows" is set to no. See Chapter 3 for details.

  Enter a value between 0 and 1. 0 indicates that the top of the widget should anchor at the top of the widget's uppermost cell. 1 indicates that the bottom of the widget should anchor at the bottom of its lowermost cell. Decimal values between 0 and 1 indicate that the widget should align proportionally between its top and bottom cells. The default is .5, or center alignment.

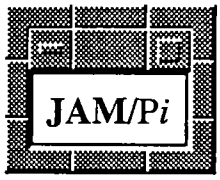
- Adjust rows (`noadj`)

  Set this option to no if you wish the positioning algorithm to ignore this widget in its vertical calculations. This is useful for tall widgets that have ample whitespace above or below them, since it prevents them from stretching the grid. It is often used in conjunction with the vertical align-

ment option, which controls where a non–adjusted widget anchors (see `valign` above). This option defaults to yes.

- Adjust columns (`noadj`)

  Set this option to no if you wish the positioning algorithm to ignore this widget in its horizontal calculations. Since the positioning algorithm uses up horizontal whitespace before stretching the grid, this option is of limited use, since it tends to cause widgets to overlap. This option defaults to yes.



Chapter 6
Extension Reference

Field and screen extensions provide access to the multitude of features available under GUI's. Here the developer may specify fonts, colors, window decorations, positioning, and specialized widgets. This chapter is a reference for the extensions, Chapter 5 explains how to enter them into the formatted screens provided with **JAM/P*i***.

6.1
INTRODUCTION

Extensions are stored in the JPL modules associated with fields and screens:

- Field extensions are stored in the field level JPL module. Their scope is the widget that represents the field.
- Screen extensions are stored in the screen level JPL module. Their scope is the screen on which they appear.

Extensions may be entered directly into the JPL comments, or they may be entered through a set of formatted screens described in Chapter 5.

Certain options that may be set via the extensions may also be specified as application defaults in the resource or initialization file. These are discussed in Chapter 7. Hierarchically, field extensions override screen extensions, which in turn override the resource or initialization file.

M In P*i*/Motif, if you specify a resource specific to a widget or class of widgets, you can override the screen or field extensions. For example, the resource setting:

```
XJam*XmText*fontList:       -*-courier-bold-r--24-*
```

changes the font of all text widgets. Resources may also be restricted to a widget, to a screen or to a class of widgets on a screen. Refer to Chapter 7 for details.

6.2 EXTENSION SYNTAX

Field and screen extensions are specified in the JPL module comments. Comments in JPL begin with the # character. Extensions are set off from other comments by double angle brackets (pairs of "less than" and "greater than" signs), as in:

```
# comment text
# <<extension(arguments)>> comment text
```

Since extensions are in the comments, they are not part of the executed JPL. This makes applications that use extensions portable to environments that don't support the extensions: a special parser interprets the extensions in **JAM/Pi**, but they are simply ignored in character **JAM**.

The parser looks only as far as the first non–comment line in each JPL module, so extensions *must* appear at the top of the module, before any blank lines or JPL code. Comments may appear on the same line as extensions, and more than one extension may appear on a line. Text lines in JPL are limited to 254 characters. Extensions that are specified incorrectly are ignored by the parser.

NOTE: Currently, no syntactic error checking is performed on the extensions. Rather than entering extensions directly into the JPL module, it is easier and more convenient to enter extensions into the formatted screens that are accessed via the SPF11 and SPF12 keys. When these screens are processed, the extensions are written into the JPL, and the developer is guaranteed that the syntax is correct.

6.2.1 Colon Expansion of Extension Arguments

Arguments to screen and field extensions are colon expanded before they are processed. Colon expansion occurs when **JAM/Pi** is about to open the GUI window to display the screen. At this point, the screen entry function has already been called, so variables for colon expansion can be set in screen entry function. Care must be taken, though, that the fields or variables upon which the expansion is based remain unchanged for the lifetime of the screen. Since rescanning may occur at arbitrary times, these variables should be left in a stable condition.

Form variables, LDB variables, and screen–local JPL variables can be used for expansion. Arguments are expanded individually, so replacement text containing commas does not create more arguments. Two examples are shown below:

```
#<<title(:mytitle)>>
#<<scale(:min,:max,:places)>>
```

6.3
PROPAGATING EXTENSIONS

Since field and screen extensions are located in JPL modules, you may use the `save to file` and `retrieve from file` functions of JPL screens, or the GUI cut and paste operations to copy extensions from one field or screen to another. The file functions are accessed via the `PF4` key from a JPL module screen. You may also use the template feature when creating a new screen to propagate extensions from one screen to another.

Propagating Fonts and Colors

The font and color screen extensions affect widgets that don't have font or color field extensions of their own. For a standardized format, you can use the font and color screen extensions once on each screen instead of using the field extensions for each field.

M **O** In P*i*/Motif and P*i*/OPEN LOOK, you can specify resources such as fonts and colors in the resource file and restrict them to a class of widgets, to a particular widget, to a screen or to a class of widgets on a screen. Refer to Chapter 7 for details.

6.4
EXTENSION REFERENCE

The following pages constitute the field and screen extension reference section. Listings appear alphabetically, but some related extensions are grouped together, specifically: foreground and background color; height and width; horizontal and vertical offset; and horizontal and vertical alignment. The two tables below indicate the page that each extension appears on, and provide a quick reference to the syntax of each extension. The first table covers field extensions, and the second covers screen extensions. The tables are organized by extension type.

NOTE: The iconification and window decoration screen extensions are implemented as hints to the window manager. This means the window manager may ignore any of these requests that it deems problematic. It can ignore any or all of them, partially or completely, although usually it does not.

Field Extensions		
Type	*Syntax*	*Page*
Incremental Positioning		
Height	`height(`***value*** `[`***units***`])`	96
Width	`width(`***value*** `[`***units***`])`	96
Horizontal Offset	`hoff(`***distance*** `[`***units***`])`	102
Vertical Offset	`voff(`***distance*** `[`***units***`])`	102
Horizontal Alignment	`halign(`***value***`)`	94
Vertical Alignment	`valign(`***value***`)`	94
Disable Adjustment	`noadj(`***direction***`)`	115
Equally Space an Array	`space(`***distance*** `[`***units***`])`	140
Fonts, Colors and Decorations		
Foreground Color	`fg(`***color***`)`	81
Background Color	`bg(`***color***`)`	81
Font	`font(`***fontname***`)`	89
Bitmapped Image	`pixmap(`***name***`)`	130
Frame	`frame ( [`***style***`,` ***color***`,` ***matte***`,` ***margin***`] )`	92
Specialized Widgets		
Checklist Toggle Button	`checkbox`	87
In/Out Toggle Button	`togglebutton`	143
Label Widget	`label`	107
List Box	`list [(no hbar, no vbar)]`	108
List of Options	`optionmenu [(`***selectscreen***`,` ***init***`,` ***popup***`)]`	127
Multiline Text Widget	`multitext [(no hbar, no vbar)]`	113

Field Extensions		
Type	*Syntax*	*Page*
Multiline Button	`multiline`	111
No Widget	`nowidget`	126
Push Button	`pushbutton`	136
Radio Toggle Button	`radiobutton`	138
Scale Widget	`scale(`***min, max***`)`	139
Text Widget	`text`	141

Screen Extensions		
Type	*Syntax*	*Page*
Fonts and Colors		
Font	`font(`***fontname***`)`	89
Foreground Color	`fg(`***color***`)`	81
Background Color	`bg(`***color***`)`	81
Lines and Boxes		
Horizontal Line	`hline(`***r, c1, c2*** `[, ` ***style, color, width, margin*** `])`	98
Vertical Line	`vline(`***c, r1, r2*** `[, ` ***style, color, width, margin*** `])`	98
Box	`box(`***l1, c1, l2, c2*** `[, ` ***style, color, matte, margin*** `])`	84
Screen Behavior		
Associate Icon with Screen and Allow Iconification	`icon(`***name***`)`	104
Start the Screen as an Icon	`iconify`	106
Specify the Pointer Shape	`pointer(`***cursor***`)`	134
Window Decorations and Features		
Suppress GUI Border	`noborder`	116
Suppress GUI Window Menu	`nomenu`	120
Disable Resize	`noresize`	124
Disable Maximize	`nomaximize`	119
Disable Iconification	`nominimize`	122
Disable Move (from menu)	`nomove`	123
Disable Close (from menu)	`noclose`	118
Invoke Maximized	`maximize`	110
Create Dialog Box	`dialog`	88
Title Bar Text	`title(`***string***`)`	142
Suppress Title Bar	`notitle`	125

<<bg>>
<<fg>>
specify the background or foreground color for a screen or widget

SYNOPSIS

```
# <<fg(color)>>
# <<bg(color)>>
```

TYPE

Field Extension
Screen Extension

DESCRIPTION

JAM/*Pi* supports a palette of sixteen colors that are specified in the resource or initialization file. Sixteen colors are usually enough for an application, as too many colors make screens hard to read. If you require more than sixteen colors, the `fg` and `bg` screen and field extensions set the foreground and background colors of screens and widgets to any color that the GUI supports.

`fg` and `bg` as Field Extensions

The `fg` field extension sets the foreground color of a widget. The `bg` field extension sets the background color of a widget. These field extensions override any other color specifications that may be applicable to the widget.

W In *Pi*/Windows, the color of a push buttons cannot be changed by **JAM** unless the `multiline` extension is used. Refer to page 111.

`fg` and `bg` as Screen Extensions

The `fg` screen extension sets the color of any foreground on the screen whose attributes are `white unhighlighted` to the color specified. `white unhighlighted` is the default foreground color in the Screen Editor display attributes screen. `fg` affects both display text and fields. `fg` is provided for convenience, as it allows you to change the foreground color of many objects at once.

M O The `fg` screen extension is similar to setting the Motif or OPEN LOOK `foreground` resource, but its scope is limited to the current screen.

The `bg` screen extension sets the color of the screen background, as well as any other background on the screen that has the exact same display attributes as the screen background, to the color specified. For example, if the screen background according to the display attributes is `red highlighted`, and the screen extension says `<<bg(goldenrod>>`, then *any* background on the screen that is `red highlighted` becomes goldenrod. This extension is designed so that any object whose background matches the screen background continues to match the screen background, even when it is changed.

M O Note that this differs from the Motif or OPEN LOOK `background` resource. The `background` resource only changes black backgrounds to the color specified, and so is consistent throughout the application.

Specifying the Color

color may be either a GUI dependent color specification or a GUI independent alias.

GUI Dependent Colors

W In Pi/Windows, specify a color as an RGB (Red/Green/Blue) value in whole numbers, as in:

```
<<fg(0/0/255)>>
```

which specifies blue. You may wish to use the Windows Control Panel to select a color and then copy the values to your extension specification. Windows limits foregrounds to "primary" colors, ie—no dithered patterns. If you specify a non-primary color, Windows rounds it up to a primary color. Most PC monitors support 16 primary colors, but some support more.

M In Pi/Motif, specify a color by name. The colors available on your system are listed in the `rgb.txt` file, usually found in the `/usr/lib/X11` directory.

O In Pi/OPEN LOOK, specify a color by name. The colors available on your system are listed in the `rgb.txt` file, usually found in the `/usr/openwin/lib` directory.

GUI Independent Color Aliases

To simplify color specification, use the color aliasing feature. Color aliasing allows you to make up your own names for ***color***, like "champagne", "gun metal grey" or "Taupe", and then specify their equivalent GUI dependent values in an alias list in the resource or initialization file. For example, you might specify `<<fg(pink)>>` as a field extension. The Motif and OPEN LOOK resource files would then have an alias pair like:

```
pink = salmon \n\
```

and the Windows initialization file would have an alias pair like:

```
pink = 247/138/115
```

For instructions on creating the alias list, refer to section 7.4.

Color aliasing enhances development flexibility, since you can change color choices in one place (the initialization or resource file) and affect changes throughout the application. It also enhances portability among GUI's, since GUI independent color names are resolved externally to your application.

<<box>>

draw a box

SYNOPSIS

```
# <<box(l1, c1, l2, c2 [, style, color, matte, margin] )>>
```

TYPE

Screen Extension

DESCRIPTION

This screen extension draws a box in the rectangle described by the specified coordinates. Box style, fill color, matte width and margin size can be optionally specified. A comma must be inserted as a placeholder for any item not specified. Boxes lay behind other widgets on the screen.

l1, ***c1***, ***l2***, and ***c2*** are one–based **JAM** lines and columns. For example, `<<box(1,1,1,1,,,,)>>` draws a box around the single cell at line one, column one.

style describes the appearance of the box. It may be any one of the following keywords:

W `single dash dot dashdot dashdotdot`

`single` is the default box style in P*i*/Windows. The ***style*** keywords for Windows refer to the border of the box. The border only appears if the box has no color specification. If the box has a color, then ***style*** is ignored and the inside of the box shows up entirely in the specified color.

M `etched in etched out in out`

`etched in` is the default box style in P*i*/Motif.

O `single`

Style is ignored in P*i*/OPEN LOOK. A single pixel border is drawn around all boxes.

color is the background color of the box. It may be either a GUI dependent or GUI independent color specification. For more on colors, see page 149.

W In P*i*/Windows, if no ***color*** is specified, a transparent box is drawn, with only a border of the specified ***style***. The color of the border is chosen by JAM/P*i* so as to be visible against the background.
If a ***color*** is specified, the box is filled in, and the ***style*** argument is ignored.

M In P*i*/Motif, if no ***color*** is specified, the background color of the form is used. Since Motif uses 3–D border styles, a box with a background the same as the screen is visible.

O In P*i*/OPEN LOOK, if no color is specified, a transparent box is drawn, with a single pixel solid border. The color of the border may be set in the resource file.

matte is the width of the area between the edge of the cells and the edge of the box. It increases the size of the box beyond the edge of its cells. If you put a box around a group of fields, it looks better if there is a matte of at least 3 pixels between the fields and the box edge.

margin is the blank margin around the outside of the box. It provides a blank area between the box and any adjoining cells. It insures that other objects outside of the box don't get too close.

The value of ***matte*** or ***margin*** may be in pixels, characters, grid units, inches, or millimeters. Refer to the chart on page 96 for a list of unit suffixes.

Figure 41 illustrates two screens with the boxes. The first has no matte or margin, and the second has both a matte and a margin.

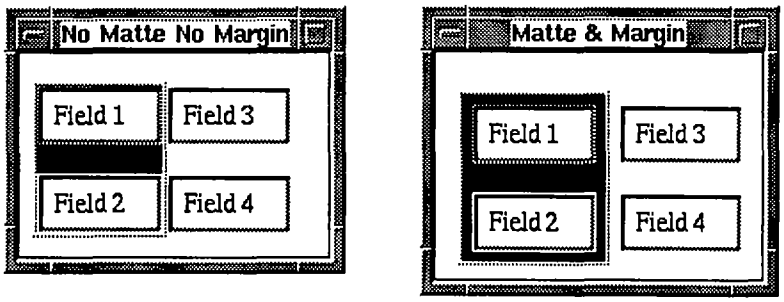


Figure 41: Two screens with black boxes. The box around fields 1 and 2 on the left hand screen has no matte or margins. The box on the right hand screen has a 5 pixel matte and a 5 pixel margin.

Figure 42, below, illustrates the parts of boxes, and how boxes affect the elastic grid. Lines and box edges are drawn in special "separator rows" and "separator columns" that appear between regular rows and columns. Separator rows and columns are just wide enough to accommodate their contents.

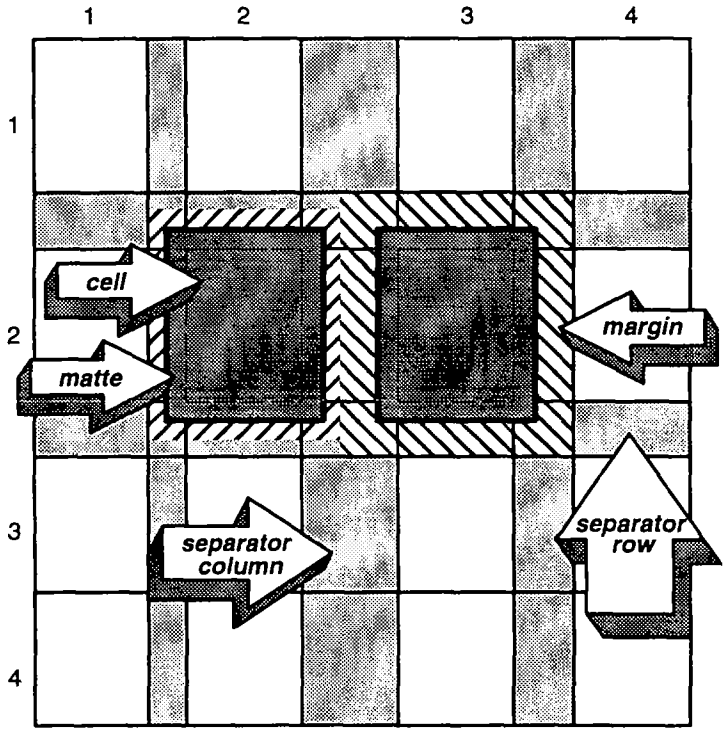


Figure 42: Two one–cell boxes that have different margins. The edges of the boxes are drawn in separator rows and columns that are just wide enough to accommodate the matte, lines and margins.

In locations where lines and boxes cross each other or overlap, the order that they appear in the screen level JPL module determines how they are layered. The first extension encountered in the module is the top–most object. The next object defined in the module is layered beneath the first object, and so on.

RELATED EXTENSIONS

```
# <<frame[ (style, color, matte, margin) ]>>
# <<hline(r, c1, c2 [, style, color, width, margin] )>>
# <<vline(c, r1, r2 [, style, color, width, margin] )>>
```

<<checkbox>>

create a checklist style toggle button

SYNOPSIS

```
# <<checkbox>>
```

TYPE

Field Extension

DESCRIPTION

This extension creates a checklist style toggle button from a field. Members of checklist groups default to this widget type. To function properly the field must be a member of a checklist group. If it is not, the developer must add callbacks to handle selection processing. This is not recommended.

M * O * In Pi/Motif and Pi/OPEN LOOK, only checklists with boxes become checklist style toggle buttons. Checklists without boxes become in/out style toggle buttons. You can use this extension to create a checklist style button from a checklist field without boxes. To avoid confusing the end-user, the `checkbox` extension should be applied to each member of the checklist group.

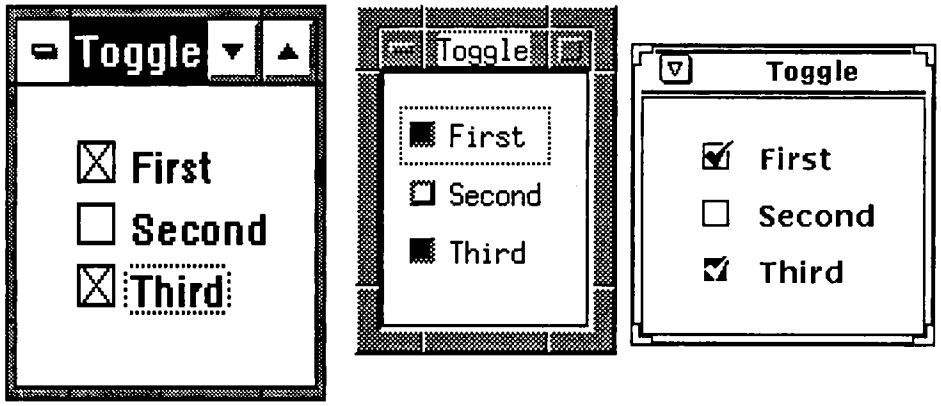


Figure 43: Checklist style toggle buttons in Windows, Motif and OPEN LOOK.

<<dialog>>

create a dialog box from a screen

SYNOPSIS

```
# <<dialog>>
```

TYPE

Screen Extension

DESCRIPTION

This extension makes a screen into a dialog box. A dialog box is an application modal window that cannot be resized, maximized or minimized.

In Pi/Windows, dialog boxes are not restricted by the MDI frame. They are free to move anywhere on the display. When a dialog box is open, screens in the MDI frame cannot be moved or resized. **JAM/Pi** dialog boxes use the style of standard MS Windows dialog boxes.

Since it is modal, the user is forced to deal with a dialog box before continuing with the application. A screen with the `dialog` extension may not be sibling, it will always be application modal. Only another dialog box can be opened on top of a displayed dialog box. If a window without the `dialog` extension opens on top of a dialog box, **JAM/Pi** forces that window to be a dialog box too.

The `noborder`, and `iconify` screen extensions are ineffective in a dialog box, and any viewport size specifications are ignored when a dialog box opens.

NOTE: The developer must not use `wselect` to give focus to a window below a dialog box that is not itself a dialog box. Doing so is undefined.

NOTE: This extension is not supported in Pi/Motif or Pi/OPEN LOOK.

<<font>>

specify the font for a screen or widget

SYNOPSIS

```
# <<font(fontname)>>
```

TYPE

Field Extension
Screen Extension

DESCRIPTION

The `font` screen extension specifies the default font for a screen. The `font` field extension specifies the font for a particular widget.

Fonts may be specified at several levels:

1. The application default font is specified in the resource or initialization file, or on the Motif command line. If a font is specified on the command line, it overrides the one specified in the resource file. In the absence of any other font specification, the application default font will be the font used for the entire application.

2. The default screen font is either the application default font or a font specified with the `font` screen extension. A `font` screen extension overrides the application default font. In the absence of any other specification, this font is used by all display text and widgets on the screen.

3. The widget's font is either the default screen font or a font specified with the `font` field extension. A `font` field extension overrides the default screen font. A region of display text can be made to have a widget's font by converting the display text into a protected field. See section 3.2.4.

Specifying the Font

The ***fontname*** argument to this extension can be either a GUI dependent font name or a GUI independent font alias. These are described below.

GUI Dependent Font Names

Pi/Windows uses the following font naming convention:

***fontname*-*pointsize*[-bold][-italic][-underline]**

fontname and ***pointsize*** are required values. `bold`, `italic` and `underline` are optional. For example:

```
Tms Rmn-24-bold
```

means Times Roman 24 point bold. Use the MS Windows Control Panel to find out what fonts are installed on your system.

Details on Windows font naming can be found in section 7.3.

Motif and OPEN LOOK use the XLFD font specification. XLFD fonts use the following naming convention:

-foundry-family-weight-slant-width-style-pixel size-point size-x resolution-y resolution-spacing-average width-charset registry-charset encoding

Case is ignored in the font name specification. Wildcards may be used for any of the values, but the more exact a specification is, the more likely that the correct font is selected. The following are example font specifications:

```
-adobe-helvetica-bold-r-normal--24-240-75-75-p-130-iso8859-1

*helvetica-bold-r-normal--24-240*

-*helvetica*24*
```

Motif and OPEN LOOK provide an application, `xfontsel`, to aid in locating fonts. Details of the XLFD font specification and the `xfontsel` program are described in section 7.3.

GUI Independent Font Aliases

To simplify font naming, use the aliasing feature. Font aliasing allows you to make up your own designations for ***fontname***, like "small", "medium" and "large", and then specify their equivalent GUI dependent names in an alias list in the resource or initialization file. For example, you might specify `<<font(bold)>>` as a field extension. The Motif or OPEN LOOK resource files would then have an alias pair like:

```
bold = *times-bold-r*14* \n\
```

and the Windows initialization file would have an alias pair like:

```
bold = Tms Rmn-14-bold
```

For instructions on creating the alias list, refer to section 7.4.

Font aliasing enhances development flexibility, since you can change font choices in one place (the initialization or resource file) and affect changes throughout the application. It also enhances portability among GUI's, since GUI independent font names are resolved externally to your application.

<<frame>>

create a frame around a widget

SYNOPSIS

```
# <<frame ( [style, color, matte, margin] ) >>
```

TYPE

Field Extension

DESCRIPTION

This field extension creates a frame around a widget, or if the widget is an array, around all the elements of the array. Edge style, color, matte width and margin size can be optionally specified. A comma must be inserted as a placeholder for any item not specified.

NOTE: Frames are different than boxes and lines in that they are drawn in the same grid cells as their associated widgets. A frame increases the size of a widget, and therefore can cause the grid to stretch. Boxes and lines, on the other hand, are drawn in special "separator" rows and columns. See page 84 for more on boxes, and page 98 for more on lines.

style describes the appearance of the frame. It can be any one of the following:

`single` `dash` `dot` `dashdot` `dashdotdot`

`single` is the default frame style in Pi/Windows. The ***style*** keywords for Windows refer to the border of the frame. The border only appears if the frame has no color specification. If the frame has a color, then ***style*** is ignored and the inside of the frame shows up in the specified color.

`etched in` `etched out` `in` `out`

`etched in` is the default frame style in Pi/Motif.

`single`

Style is ignored in Pi/OPEN LOOK. A single line border 2 pixels wide is drawn around all frames.

color is the background color. It may be either a GUI dependent or GUI independent color specification. For more on colors, see page 149.

W In P*i*/Windows, if no ***color*** is specified, a transparent frame is drawn with a border of the specified ***style***. The color of the border is chosen so as to be visible against the background.

If a ***color*** is specified, the frame is filled in and the ***style*** argument is ignored.

M In P*i*/Motif, if no ***color*** is specified, the background color of the form is used. Since Motif uses 3-D border styles, a frame with a background color the same as the screen's is visible.

matte is the width of the area between the edge of the widget and the edge of the frame. It increases the size of the frame beyond the edge of the widget. A frame looks better if there is a matte of at least 3 pixels between the widget and the frame border edge.

margin is the blank margin around the outside of the frame. It provides a blank area between the frame and the edge of the cell. It insures that other adjoining objects don't get too close to the frame.

The value of ***matte*** or ***margin*** may be in pixels, characters, grid units, inches, or millimeters. Refer to the chart on page 96 for a list of unit suffixes.

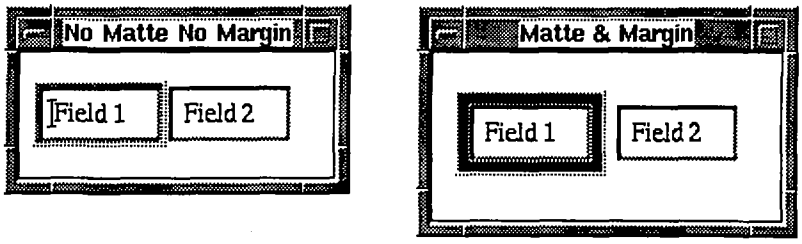


Figure 44: Two screens with a framed field. The frame around field 1 on the left hand screen has no matte or margin. The frame on the right hand screen has a 5 pixel matte and a 5 pixel margin.

RELATED EXTENSIONS

```
# <<box(l1, c1, l2, c2 [, style, color, matte, margin] )>>
# <<hline(r, c1, c2 [, style, color, width, margin] )>>
# <<vline(c, r1, r2 [, style, color, width, margin] )>>
```

<<halign>>
<<valign>>

specify an alternative horizontal or vertical alignment for this widget

SYNOPSIS

```
# <<halign(value)>>
# <<valign(value)>>
```

TYPE

Field Extension

DESCRIPTION

JAM/Pi calculates the default placement for widgets on a screen using a positioning algorithm described in Chapter 3. This algorithm takes into account many factors, including field justification, the white space available on the screen, and the size of widgets. Each widget has a certain number of rows or columns that it is supposed to occupy. These are referred to as grid cells. At times, the algorithm stretches rows or columns in order to fit large widgets onto a screen. Other widgets that span these stretched rows or columns may now be smaller than the grid cells allotted to them. **JAM/Pi** must decide where to align these objects within their allotted cells.

By default, left justified fields and display text align on their left, in their starting cell. Right justified fields align on their right, in their ending cell. The `halign` field extension enables the developer to specify any alignment for a widget, regardless of its justification.

Vertically, all widgets align by default in the center of their allotted cells. The `valign` field extension enables the developer to specify any vertical alignment for a widget.

Note that these extensions come into play only when a widget is larger or smaller than the space available in its allotted cells.

value is a number between `0` and `1`. Horizontally, `0` means that the left edge of the widget should anchor in its starting cell. `0` is the default alignment for left justified fields and display text. `1` means that the right edge of the widget should anchor in its ending (or rightmost) cell. This is the default for right justified fields. A ***value*** between `0` and `1` means that the widget should align proportionally between its starting and ending cells. Thus, `.5` means that the center of the widget should anchor in the center of the available space.

Vertically, a ***value*** of `0` means that the top of the widget should align with the top of its uppermost cell. `1` indicates that the bottom of the widget should align with the bottom of its lowermost cell. Decimal values in between indicate proportional alignment between the top and bottom cells. The default vertical alignment is `.5`, or centered.

Values for `halign` or `valign` that are less than `0` or greater than `1` result in alignment outside of the allotted cells. Alignment outside of the allotted cells may result in widgets overlapping one another. Values less than `0` or greater than `1` are *not* recommended.

Chapter 3 discusses the positioning algorithm. Read this chapter to get a full understanding of how positioning works. Figure 15 in Chapter 3 has a diagram that illustrates `halign`.

RELATED EXTENSIONS

<<hoff(***distance*** [***units***])>>

<<voff(***distance*** [***units***])>>

<<noadj(***direction***)>>

<<height>>
<<width>>

specify the width or height of a widget

SYNOPSIS

```
# <<width(value [units])>>
# <<height(value [units])>>
```

TYPE

Field Extension

DESCRIPTION

Each widget has a default size based on several factors, including the size of its font, the length or contents of its associated **JAM** object, and any widget decorations. The **JAM/Pi** positioning algorithm allocates enough screen space for a widget based on its size.

The `height` and `width` field extensions enable the developer to override the default size of a widget. Any size may be specified. The positioning algorithm uses the new size of the widget, rather than its default size, in making its calculations.

value represents the height or width of the widget. ***value*** may be either an integer, in which case it represents the height or width in pixels, or it may be any floating point number followed by the ***units*** suffix, indicating which units to used. ***units*** are listed below:

Suffix	*Units*	*Description*
p (or none)	Pixels	If no suffix is used, then the value is assumed to be in pixels. ***value*** must be an integer if it is in pixels. These measurements depend upon screen resolution.
c	Characters	A character is the average character width of the widget's font. 5c means 5 average characters in the widget's font. Contrast with grid units, which refer to the default screen font. Characters and grid units are the most portable units of measure, since they are sensitive to the font in use. (In screen extensions, characters are the same as grid units.)

Suffix	*Units*	*Description*
g	Grid Units	A grid unit is the average character width of the default screen font. 5g means 5 standard (unstretched) grid cells. Grid units and characters are the most portable units of measure, since they are sensitive to the font in use.
mm	Millimeters	The value is in millimeters. The X server must know the correct physical screen dimensions in order for these measurements to be accurate. How the server is configured, though, is machine dependent.
in	Inches	The value is in inches. The X server must know the correct physical screen dimensions in order for these measurements to be accurate. How the server is configured, though, is machine dependent.

For example, you might want to make a text widget wider if its input will be all capital letters, like a field for a state abbreviation. The default width of a widget is based on the average character width of the font times the length of the field. If the widget is using a proportional font, then an entry of all capital letters most likely won't fit, since most capital letters are wider than the average character. The user will be able to enter the correct number of characters, but they won't all display at the same time; the widget will have to scroll. If a two character field is given a `width` field extension like `<<width(3c)>>`, then any two characters are likely to display without scrolling.

Another example of when you might wish to use a `width` and a `height` field extension is to make a large (1 inch square) push button. To do this, you would simply specify the following in the field level JPL module for a menu field:

```
# <<height(1in)>> <<width(1in)>>
```

In an array with a `height` or `width` extension, each widget in the array takes on the height and width specified. So a vertical array with three elements that has a `<<height(1in)>>` extension occupies at least three inches, since it contains three widgets. Arrays with the `multitext`, `optionmenu` or `list` extensions should not have the `height` extension.

Fields with pixmaps or bitmaps respect `height` and `width` extensions.

W In Pi/Windows, bitmaps are scaled to fit in the height and width specified.

M **O** In Pi/Motif and Pi/OPEN LOOK, bitmaps and pixmaps are truncated if they don't fit in the height and width specified.

<<hline>>
<<vline>>
create a vertical or horizontal line

SYNOPSIS

```
# <<hline(r, c1, c2 [, style, color, width, margin] )>>
# <<vline(c, r1, r2 [, style, color, width, margin] )>>
```

TYPE

Screen Extension

DESCRIPTION

These screen extensions draw vertical and horizontal lines between the specified coordinates. Style, color, width and margin for lines can be optionally specified. A comma must be inserted as a placeholder for any item not specified.

Figure 45 illustrates horizontal and vertical lines in Windows and Motif.

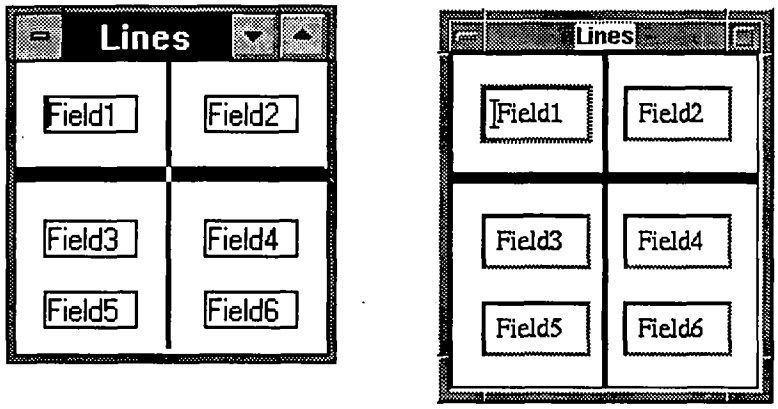


Figure 45: Screens broken into quadrants by horizontal and vertical lines.

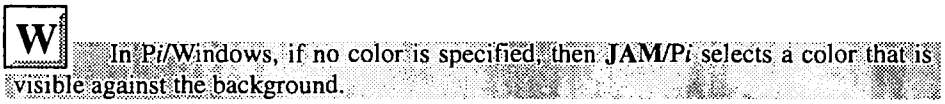
For a horizontal line, specify a row, ***r***, and a starting and ending column, ***c1*** and ***c2***. For a vertical line, specify a column, ***c***, and a starting and ending row, ***r1*** and ***r2***. Horizontal lines are drawn at the *top* of the row specified, from the left side of column ***c1*** to the right side of column ***c2***. Vertical lines are drawn at the *left* of the column specified, from the top of row ***r1*** to the bottom of row ***r2***.

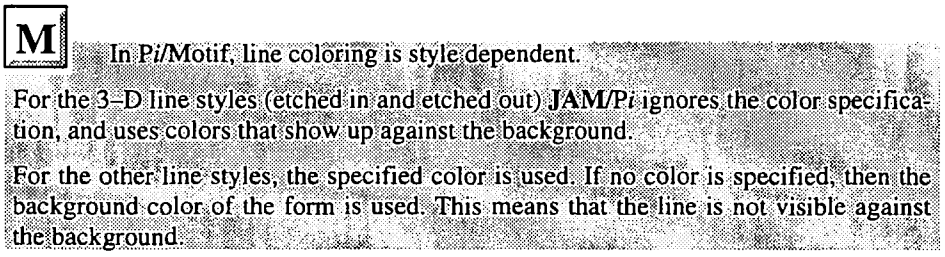
To draw a line to the right of the last column on the screen or below the last row, specify a row or column that is one greater than the last row or column. For example, on a 23x80 screen, `<<vline(81,...)>>` draws a vertical line to the right of column 80.

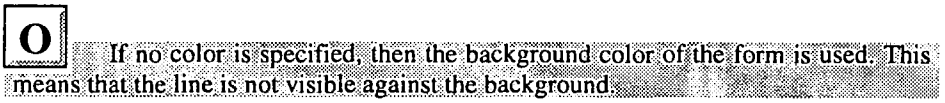
style describes the appearance of the line. It can be any one of the following:

The `single` and `dash` styles happen to be portable between Windows, Motif and OPEN LOOK. If the specified style is not supported under the GUI, a closely matching style, or the default, `single`, is used.

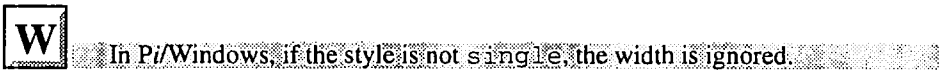
color is the color of the line. It may be either a GUI dependent or GUI independent color specification. For more on colors, refer to page 149.







width specifies the width of a line. provided that the style is `single`.



M O In Pi/Motif and Pi/OPEN LOOK, if the style is not `single`, the line is drawn in the center of the specified width.

margin specifies the size of a blank margin area on either side of the line. The value of ***width*** or ***margin*** may be in pixels, characters, grid units, inches, or millimeters. Refer to the chart on page 96 for a list of unit suffixes. ***width*** defaults to one pixel. ***margin*** defaults to zero.

Lines are drawn in "separator rows" and "separator columns" that run between grid cells. Separator rows and columns are just wide enough to hold their contents. Therefore, the width of a separator row is determined by the width of the widest line in the row and its margins, plus the matte width and margins of any box edges in the row. The same rule is true for columns. For more on boxes, see page 84.

Figure 46 illustrates where lines are drawn, and how they affect the grid.

A widget that in Draw Mode crosses a row or column containing a line, will overlap the line in Test and Application Modes. A widget that in Draw Mode does not cross the row or column boundary containing a line, will not overlap the line. Instead, the grid will stretch if necessary. For example, in the above diagram, imagine a widget in row 3 that spans columns 1 and 2. Regardless of how wide the two column widget becomes, it will not cross the vertical line in column 3. On the other hand, a widget spanning columns 1, 2, *and* 3 will overlap the line, and the line will be drawn behind the widget. The determining factor as to whether a widget overlaps a line is whether the widget crosses the row or column containing the line in *Draw Mode*. The same rule applies for the edges of boxes.

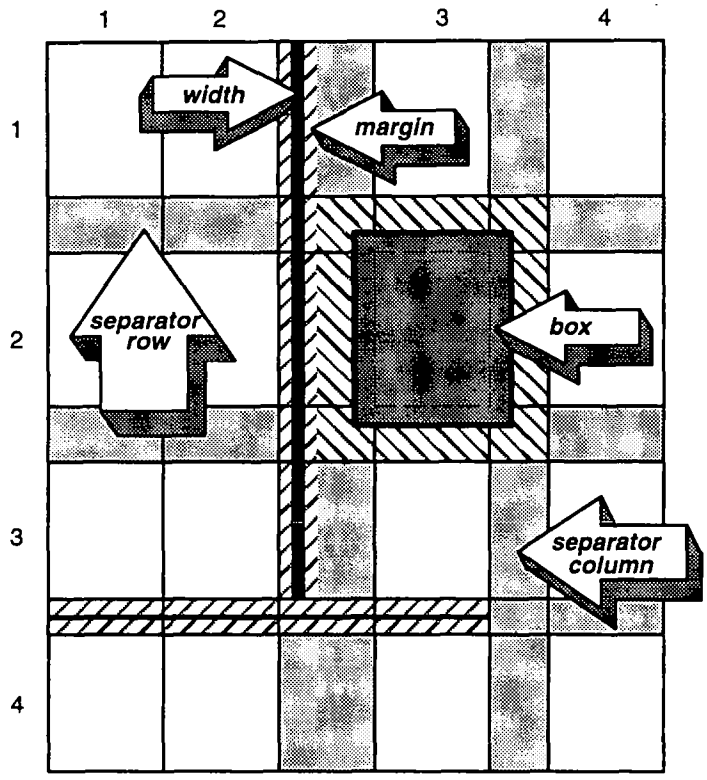


Figure 46: Screen containing two lines and a box. The vertical line is specified for column 3, the horizontal line for column 4. Lines and boxes are drawn in separator rows and columns that are sized just wide enough for them.

In locations where lines and boxes cross each other or overlap, the order that they appear in the screen level JPL module determines how they are layered. The first extension encountered in the module is the top–most object. The next object defined in the module is layered beneath the first object, and so on.

RELATED EXTENSIONS

```
# <<box(l1, c1, l2, c2 [, style, color, matte, margin] )>>
# <<frame ( [style, color, matte, margin] ) >>
```

<<hoff>>
<<voff>>

specify a horizontal or vertical offset for a widget

SYNOPSIS

```
# <<hoff(distance [units])>>
# <<voff(distance [units])>>
```

TYPE

Field Extension

DESCRIPTION

JAM/Pi calculates the default placement for widgets on a screen using a positioning algorithm described in Chapter 3. This algorithm takes into account many factors, including field justification, the white space available on the screen, and the size of widgets.

The `hoff` and `voff` field extensions move a widget a specified distance from its default position. `hoff` moves a widget horizontally. `voff` moves it vertically. These field extensions are applied after the positioning algorithm makes its calculations, so there is no guarantee that widgets with an `hoff` or `voff` will not overlap other widgets. Use these extensions sparingly, as too many `hoff` and `voff` extensions make a screen hard to maintain.

distance indicates the distance to move. A signed ***distance*** indicates movement relative to the widget's default position. An unsigned ***distance*** indicates an absolute location relative to the top or left margin.

A positive ***distance*** for `hoff` moves the widget to the right. A negative ***distance*** moves it to the left. An unsigned ***distance*** places the widget relative to the left margin.

A positive ***distance*** for `voff` moves the widget down. A negative ***distance*** moves it up. An unsigned ***distance*** places the widget relative to the top margin.

distance may be either an integer, in which case it represents the distance in pixels, or it may be any floating point number followed by a ***units*** suffix. ***units*** may be characters, grid units, inches, or millimeters. Refer to the chart on page 96 for details.

A common use of `hoff` is to obtain equal horizontal spacing between a set of objects when some large object above them on the screen has stretched the grid. Figure 47 illustrates such a screen.

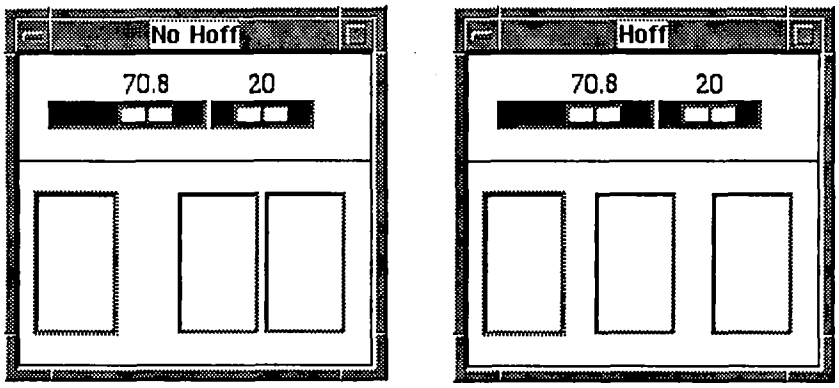


Figure 47: Screens with two scale widgets and three multiline text widgets. In the left hand screen, an oversized scale widget at top left has stretched the grid, causing unequal spacing between the widgets below it. In the right hand screen, an `hoff` screen extension on the middle multiline widget takes care of the problem.

Figure 47 illustrates a use of relative offset. An alternative solution to the unequal spacing of the widgets is absolute `hoff` extensions on each of the three multiline widgets. For example, `<<hoff(1g)>>` for the leftmost widget, `<<hoff(6g)>>` for the middle widget, and `<<hoff(11g)>>` for the rightmost widget. This places each widget in a specific location relative to the left edge of the screen. With this model, you can control exactly where each item on a screen is located.

RELATED EXTENSIONS

`<<halign(`***value***`)>>`
`<<valign(`***value***`)>>`

<<icon>>

enable iconification and associate an icon with a screen

SYNOPSIS

```
# <<icon(name)>>
```

TYPE

Screen Extension

DESCRIPTION

This extension associates the icon specified by ***name*** with a screen. A screen with the `icon` screen extension may be iconified (minimized) individually. A minimize push button appears in the screen border, and the `minimize` option is enabled on the GUI window menu. If the specified icon bitmap is not found, the default bitmap is used instead.

In Pi/Windows, any base form or sibling window can be iconified individually, regardless of whether it has the `icon` screen extension. Stacked windows cannot be minimized (see page 43). If a screen doesn't have the `icon` screen extension, then the default icon is used when the screen is minimized.

All icons used in a **JAM** application must be listed in the Windows resource file for the application. For the **JAM** authoring tool, this file is called `wjxform.rc`. The syntax in the resource file is:

name `ICON` ***filename***

where ***name*** is the name of the icon and ***filename*** identifies the disk file containing the icon. Be sure to compile the resource file and link it with the application after making any changes. Refer to your MS Windows SDK documentation for more information on resource files.

Motif icons are searched for in the directory pointed to by the `bitmapDirectory` resource. This defaults to `/usr/include/X11/bitmaps`.

OPEN LOOK icons are searched for in the `$OPENWINHOME/include/X11/bitmaps` directory and in `$HOME/bitmaps`.

RELATED EXTENSIONS

```
# <<iconify>>
# <<nominimize>>
```

<<iconify>>

start this screen as an icon

SYNOPSIS

```
# <<iconify>>
```

TYPE

Screen Extension

DESCRIPTION

This screen extension specifies that the screen should initially display in an iconified state. If the screen does not have an `icon` screen extension specified, then the default icon is used.

RELATED EXTENSIONS

```
# <<icon(name)>>
```

<<label>>

create a label widget

SYNOPSIS

```
# <<label>>
```

TYPE

Field Extension

DESCRIPTION

This extension creates a label widget from a field. Fields protected from data entry and tabbing default to this widget type. If you use this extension for a field that is not protected from data entry or tabbing, **JAM** allows tabbing and data entry in the widget, but the user does not see the cursor in the widget. This is very confusing to the user. We strongly recommend against using this extension on unprotected fields.

Label widgets for left justified fields anchor by default on their left. Label widgets for right justified fields anchor by default on their right. The `halign` extension can be used to change the default alignment. See Chapter 3 for details on the positioning algorithm used in **JAM/P*i***.

W In P*i*/Windows display text does not become a label widget. Instead, it is simply text painted on the screen.

Figure 48 illustrates label widgets.

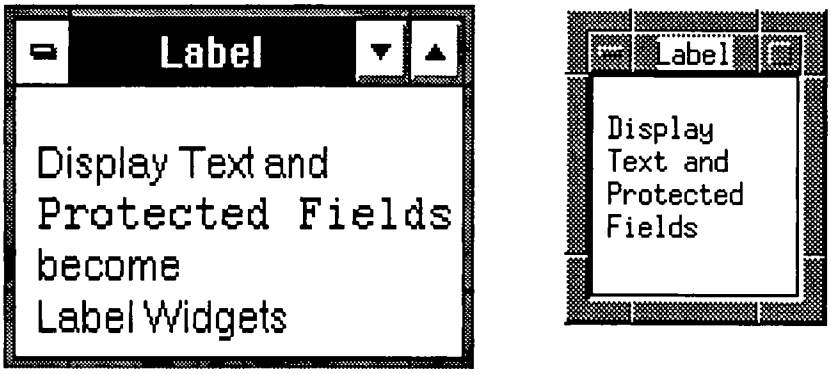


Figure 48: Label widgets in Windows and Motif.

<<list>>

create a list box from an array

SYNOPSIS

```
# <<list [(no hbar, no vbar)] >>
```

TYPE

Field Extension

DESCRIPTION

An array in **JAM/***Pi* normally consists of one widget for each element in the array. This extension transforms an array into a single widget called a list box. Items in a list box can be selected, so they are appropriate only for checklists, radio buttons and menus on item selection screens.

NOTE: Fields that are not selection criteria may be made into list boxes, but the developer must add callbacks to handle the selection event. Otherwise, the widget will look like a list box, but no selection can take place because data entry fields have no selection semantics.

Normally, items in a list box are protected from data entry and clearing, as they are selection criteria, rather than data entry fields. A radio button converted to a list box allows only one item to be selected. A checklist converted to a list box allows multiple items to be selected. Selected items appear in reverse video. Item selection screens that contain list boxes copy the selection to the underlying screen.

List boxes can be tailored to your preference for scroll bars. If no parentheses appear after the `list` keyword, then the list box has scroll bars only when appropriate. A scrolling array has a vertical scroll bar. A shifting array has a horizontal scroll bar. A shifting and scrolling array has both scroll bars.

If parentheses appear after the `list` keyword, then the list box has the specified level of scroll bar turned off, regardless of need. For example, a `list(no hbar)` widget has no horizontal scroll bar, but it always has a vertical scroll bar. A `list()` widget has both scroll bars, whether they are needed or not. If scroll bars are turned off, the widget may still be shifted or scrolled by dragging the mouse cursor beyond the edge of the widget in the desired direction, or with the **JAM** shift, scroll, or zoom keys.

NOTE: The settings regarding horizontal and vertical scroll bars are implemented as hints to the window manager. Therefore they may be ignored under certain conditions. For example in Windows 3.1, `no vbar` is ignored unless you also specify `no hbar`.

A list box anchors vertically in the center of the area available for the array it replaces. To make it anchor at the top of that area, give it a `valign` of `0`.

Single widgets that represent **JAM** arrays can have only one foreground and one background color. This means that the library routines that alter display attributes for elements or occurrences of arrays (the `_e_`, `_i_` and `_o_` variants of `sm_achg` and `sm_chg_attr`) have no effect on list boxes.

Figure 49 illustrates list boxes in Windows and Motif.

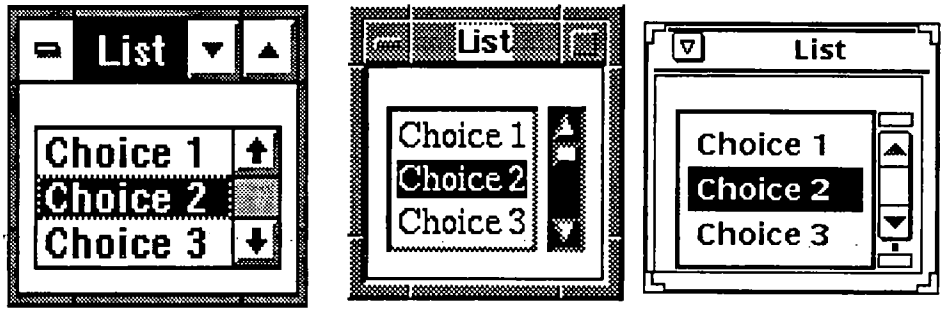


Figure 49: List boxes with vertical scroll bars in Windows, Motif and OPEN LOOK.

RELATED EXTENSIONS

```
# <<multitext [(no hbar, no vbar)] >>
# <<<optionmenu [(selectscreen, init, popup)] >>
```

<<maximize>>

invoke a window maximized

SYNOPSIS

```
# <<maximize>>
```

TYPE

Screen Extension

DESCRIPTION

This extension causes a **JAM** screen to appear in a maximized GUI window when the screen is first displayed.

W In Pi/Windows, a maximized window occupies the entire MDI frame. To bring up your application in a maximized MDI frame, use the `StartupSize` option in the application initialization file. The MDI limits the number of maximized windows to one. The maximized window must be the topmost window.

M **O** **NOTE:** This extension is not supported in Pi/Motif or Pi/OPEN LOOK.

RELATED EXTENSIONS

```
# <<nomaximize>>
# <<iconify>>
```

<<multiline>>

create a multiline label for a menu or group button

SYNOPSIS

```
# <<multiline>>
```

TYPE

Field Extension

DESCRIPTION

Certain widgets in **JAM/***Pi* have a label associated with them. These are: toggle buttons (for checklists and radio buttons), push buttons and label widgets. Normally the label has only one line of text. This extension enables the label to have multiple lines of text.

The first line of text is stored in the field's on–screen data. The subsequent lines are stored in the field's off–screen data, so if you wish to have more than one line of text, use a shifting field. The length of each text line in a multiline widget is equal to the on–screen length of the field, and the number of lines is determined by the field's shifting length. For example, a field whose on–screen length is 5 and total length is 14 will have 3 lines of text. The first five characters in the field will appear on line 1, the next five characters on line 2, and the last four on line 3.

Use the ZOOM key in draw mode to enter text into the shifting field, remembering to include sufficient spaces to make the text lines break properly.

A multiline widget occupies only one row of the grid, so it stretches the grid vertically if it contains more than one line. You may use the `noadj(rows)` field extension to prevent grid stretching for a multiline widget, as long as there is whitespace available above or below the widget. Use `valign` to align the widget vertically.

In *Pi*/Windows, only menu fields can become multiline widgets.

A side benefit of the `multiline` extension is that it allows buttons to have a different color under MS Windows. Normally, Windows restricts the color of buttons to one choice that is made in the `win.ini` file. Multiline buttons with an unhighlighted white foreground or an unhighlighted black background use the colors from `win.ini`. But if a multiline button that has a different foreground or background color, that color is used instead. To change the color of a single line button, give it the `multiline` extension, but don't use a shifting field. Be aware that a multiline button may not look like a button depending on the color choices you make.

O In Pi/OPEN LOOK, only labels can become multiline widgets. Toggle buttons and push buttons ignore this extension.

Figure 50 illustrates a multiline button in Motif and Windows. Follow the following to steps create this button:

1. Create a field of length 8
2. Give the field a shifting length of 24
3. Protect the field from data entry and tabbing.
4. Give the field the menu edit.
5. Give the field the `multiline` extension.
6. Enter the following text into the field:

```
A Buttonwith 3    lines
```

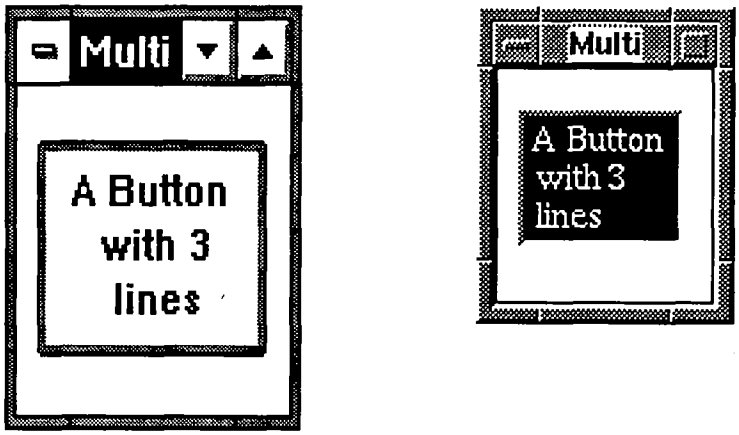


Figure 50: A multiline button in Windows (left) and Motif (right).

<<multitext>>

create a multiline text widget from an array

SYNOPSIS

```
# <<multitext [(no hbar, no vbar)] >>
```

TYPE

Field Extension

DESCRIPTION

An array in **JAM/P*i*** normally consists of one text widget for each element in the array. This extension transforms an array into a multi–line text widget. A multi–line text widget is like a regular text widget, except that it has as many text lines as the array has on–screen elements, all enclosed in the same border. Multi–line text widgets are appropriate for both word wrap arrays and arrays containing discrete data elements. They are not appropriate for groups or menus.

Multiline text widgets can be tailored to your preference for scroll bars. If no parentheses appear after the `multitext` keyword, then the array has scroll bars only when it is appropriate. A scrolling array has a vertical scroll bar. A shifting array has a horizontal scroll bar. A shifting and scrolling array has both scroll bars.

If parentheses appear after the `multitext` keyword, then the widget has the specified level of scroll bar turned off, regardless of need. For example, a `multitext(no hbar)` widget has no horizontal scroll bar, but always has a vertical scroll bar. A `multitext()` widget has both scroll bars, whether they are needed or not.

If scroll bars are turned off, the widget may still be shifted or scrolled by dragging the mouse cursor beyond the edge of the widget in the desired direction, or with the shift, scroll or zoom keys.

NOTE: The settings regarding horizontal and vertical scroll bars are implemented as hints to the window manager. Therefore they may be ignored under certain conditions. For example, all multitext widgets in OPEN LOOK have scroll bars.

Figure 51 illustrates how a multiline text widget appears, as opposed to a regular array, in Windows and Motif.

Single widgets that represent **JAM** arrays can have only one foreground and one background color. This means that the library routines that alter display attributes for elements or occurrences of arrays (the `_e_`, `_i_` and `_o_` variants of `sm_achg` and `sm_chg_attr`) have no effect on list boxes.

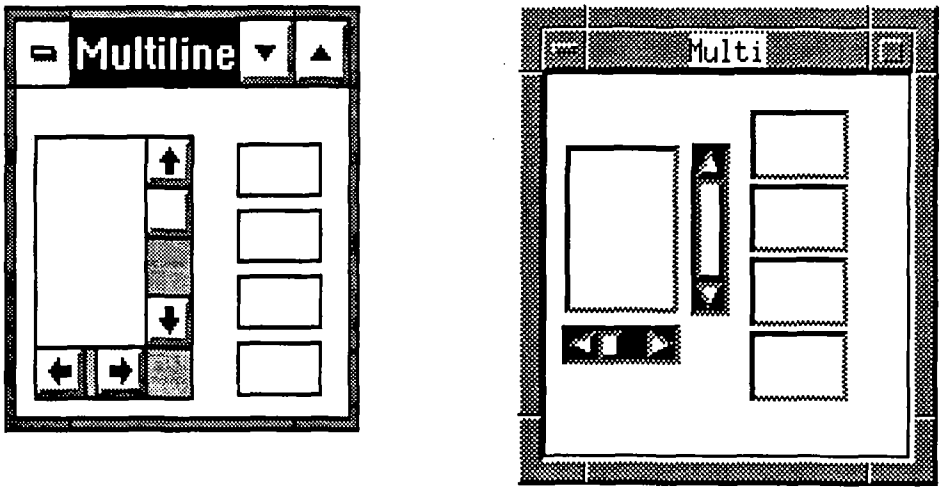


Figure 51: Multiline text widgets versus regular arrays in Windows and Motif.

RELATED EXTENSIONS

```
# <<list [(no hbar, no vbar)] >>
```

<<noadj>>

disable vertical or horizontal grid adjustment for a widget

SYNOPSIS

```
# <<noadj(direction)>>
```

TYPE

Field Extension

DESCRIPTION

JAM/P*i* calculates the default placement for widgets on a screen using a positioning algorithm described in Chapter 3. This algorithm takes into account many factors, including field justification, the white space available on the screen, and the size of widgets. Each widget occupies a certain number of rows or columns, referred to as grid cells. At times, the algorithm stretches rows or columns in order to fit large widgets onto a screen.

The `noadj` field extension indicates that a widget should not be considered by the positioning algorithm in its calculations. As a result, the elastic grid does not stretch to accommodate the widget. This means that if the widget is large, it may run over into cells that it would not normally occupy, whether those cells are occupied by another widget or not. Thus, `noadj` can result in widgets overlapping each other or clipping the edge of the screen.

direction may be either the literal word `rows` or `columns`. `noadj(rows)` turns off vertical grid adjustment, and `noadj(columns)` turns off horizontal grid adjustment.

This extension is mostly used in the vertical direction for tall widgets that have space available above or below them. `noadj(rows)` prevents the tall widget from distorting the vertical alignment of other widgets that happen to lie in the same rows. You may wish to use `valign` in combination with `noadj`, to control where a widget aligns vertically. See page 94 for more on `valign`.

`noadj` is less useful horizontally, since the default behavior of the positioning algorithm is to use up available whitespace around a widget before stretching the grid. `noadj(columns)` simply tends to make widgets overlap.

See Figure 21 on page 35 for an example of `noadj`. Refer to Chapter 3 for more on the positioning algorithm.

RELATED EXTENSIONS

```
# <<halign(value)>>
# <<valign(value)>>
```

<<noborder>>

suppress the GUI border for this screen

SYNOPSIS

```
# <<noborder>>
```

TYPE

Screen Extension

DESCRIPTION

The GUI windows that contain **JAM** screens are normally drawn with a GUI border and resize handles. The `noborder` screen extension suppresses the border and resize handles, leaving only a bounding box.

M In P*i*/Motif, this extension also removes the title bar and the minimize, maximize and GUI window menu buttons. As a result, `noborder` screens cannot be moved, resized, minimized or maximized with the mouse by the end user. GUI keyboard shortcuts can still perform these functions, though.

O In P*i*/OPEN LOOK, this extension also removes the title bar and the minimize, maximize and GUI window menu buttons. As a result, `noborder` screens cannot be resized with the mouse by the end user.

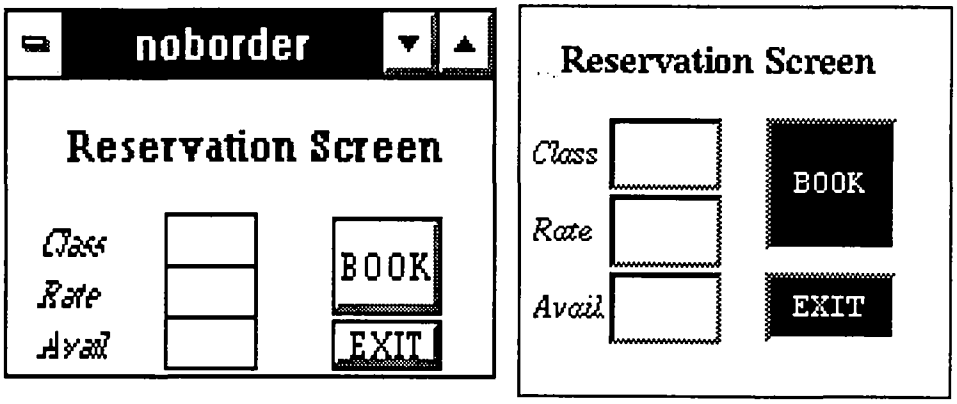


Figure 52: `noborder` screens in P*i*/Windows and P*i*/Motif.

RELATED EXTENSIONS

```
# <<notitle>>
```

<<noclose>>

suppress the close option on the GUI window menu

SYNOPSIS

```
# <<noclose>>
```

TYPE

Screen Extension

DESCRIPTION

This screen extension suppresses the close option on the GUI window menu. This prevents the user from closing the window via the mouse.

NOTE: This extension is not supported in Pi/OPEN LOOK.

RELATED EXTENSIONS

```
# <<nomenu>>
```

<<nomaximize>>

prevent the user from maximizing a window

SYNOPSIS

```
# <<nomaximize>>
```

TYPE

Screen Extension

DESCRIPTION

GUI windows usually have a maximize button in their border. This screen extension removes the maximize button from the title bar and the maximize entry from the GUI window menu. This prevents the user from maximizing the window.

NOTE: This extension is not supported in P*i*/OPEN LOOK. Use `noresize` to prevent the user from enlarging the window.

RELATED EXTENSIONS

```
# <<nomenu>>
```

<<nomenu>>

suppress the GUI window menu

SYNOPSIS

```
# <<nomenu>>
```

TYPE

Screen Extension

DESCRIPTION

Each GUI window has a "window menu" with options on it for controlling various aspects of the window. This menu is accessed by a button that appears in the upper left hand corner of the GUI window's border. Items on the window menu depend upon the GUI, but usually include: Restore, Move, Size, Minimize, Maximize and Close. Most features on this menu also have other means of access, such as resize handles, the maximize button, or keyboard shortcuts. The `nomenu` screen extension suppresses the window menu button, and prevents the user from accessing the menu. It does not inhibit the features listed on the menu if they are accessible through another means.

In Pi/Windows, `nomenu` implies `nominimize` and `nomaximize`.

In Pi/OPEN LOOK, this extension removes the menu button, but may not prevent the user from bringing up the menu.

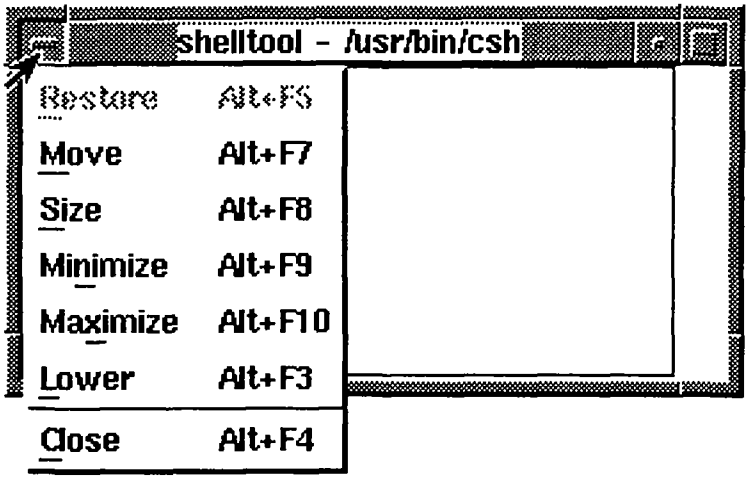


Figure 53: The GUI window menu in Motif.

<<nominimize>>

prevent the user from minimizing a GUI window

SYNOPSIS

```
# <<nominimize>>
```

TYPE

Screen Extension

DESCRIPTION

This screen extension prevents the user from minimizing a screen by removing the minimize button from the border, and removing the minimize entry from the GUI window menu.

M In Pi/Motif, only screens that have the `icon` screen extension may be minimized by the user. The `nominimize` extension is therefore rarely used.

O **NOTE:** This extension is not supported in Pi/OPEN LOOK.

RELATED EXTENSIONS

```
# <<icon(name)>>
# <<nomenu>>
```

<<nomove>>

suppress the move option on the GUI window menu

SYNOPSIS

```
# <<nomove>>
```

TYPE

Screen Extension

DESCRIPTION

This screen extension suppresses the move option on the GUI window menu. It does not however suppress the move handle on the GUI window, so the window may still be repositioned by the user, unless the `noborder` or `notitle` extension is used as well.

NOTE: This extension is not supported in Pi/OPEN LOOK.

RELATED EXTENSIONS

```
# <<noborder>>
# <<nomenu>>
# <<notitle>>
```

<<noresize>>

prevent the user from resizing a GUI window

SYNOPSIS

```
# <<noresize>>
```

TYPE

Screen Extension

DESCRIPTION

GUI windows containing **JAM** screens are normally drawn with resize handles in the window border. The `noresize` screen extension suppresses these handles, and removes the "size" option from the GUI window menu. The user will no longer be able to shrink or expand such a window. Since the window has no resize handles, the border will be slightly narrower than normal. Figure 54 compares a window with resize handles to one without resize handles.

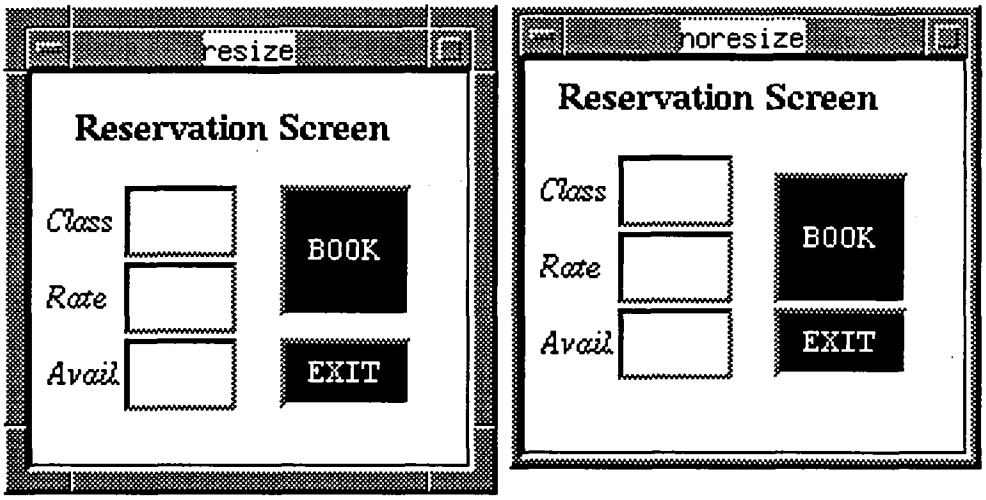


Figure 54: A Motif screen with and without resize handles.

RELATED EXTENSIONS

```
# <<noborder>>
```

<<notitle>>

suppress title bar

SYNOPSIS

```
# <<notitle>>
```

TYPE

Screen Extension

DESCRIPTION

GUI windows normally have a title bar. This extension suppresses the title bar and the decorations on it: the minimize, maximize and GUI window menu buttons. This is illustrated in Figure 55.

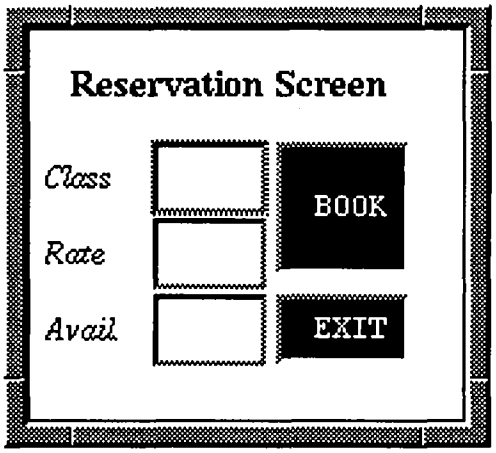


Figure 55: A screen with the `notitle` extension. It has no title bar, minimize button, maximize button or GUI window menu button.

If you wish to suppress only the text in the title bar, use the extension `title()`. See page 142 for details.

RELATED EXTENSIONS

```
# <<title(string)>>
# <<noborder>>
```

<<nowidget>>

don't create a GUI widget for this field

SYNOPSIS

```
# <<nowidget>>
```

TYPE

Field Extension

DESCRIPTION

This extension prevents a widget from being created for this field. Protected fields that are non–display (such as menu control fields) default to this widget type.

In terms of positioning, a `nowidget` field occupies the number of columns that the field was drawn in. These columns are not considered whitespace, even though they contain no GUI objects. This means that other widgets on the screen are not free to expand into the area that a `nowidget` field occupies.

<<optionmenu>>

create an option menu widget

SYNOPSIS

```
# <<optionmenu [(selectscreen, init, popup)] >>
```

TYPE

Field Extension

DESCRIPTION

An option menu widget allows the user to pull up a list of options and choose one. The user clicks on an indicator in the widget to pop up the list of options, or uses the arrow keys to scroll through them. There are two variations of optionmenus. In the first variation, the list of options is contained in the off–screen occurrences of the field. In the second, the list of options comes from another screen, much like item selection screen.

In the first variation, the `optionmenu` extension is specified without arguments. This converts a scrolling array into an option menu widget. The underlying array should:

- have one element.
- have as many occurrences as there are options in the list.
- be protected from data entry and clearing.
- *not* be protected from tabbing.
- be circular.

The initial data in the occurrences of the array make up the items in the option menu. In the character world, this is sometimes called a cycle field, because the user can tab to the field and cycle through the choices with the arrow keys. Use the library routine `sm_e_getfield` to determine the user's selection.

The first occurrence in the array is the default value in the field. If you want the field to default to blank, add an extra occurrence to the array, and make the first occurrence blank.

Single widgets that represent **JAM** arrays can have only one foreground and one background color. This means that the library routines that alter display attributes for occurrences of an array (`sm_i_achg` and `sm_o_achg`) have no effect on option menus made from cycle fields.

In the second variation, the `optionmenu` extension is specified with a ***selectscreen*** argument. This indicates that the values in the pop–up should be retrieved from another screen, much like an item selection screen.

A **JAM** field with this variation of `optionmenu` should be a non–scrolling field or array. Each array element gets its own optionmenu widget. If you wish the user to select only from the list of choices on an optionmenu, protect the field from data entry and clearing. If the field is not protected from data entry, the user may type directly into the `optionmenu` widget. This allows the widget to function like a Windows combo box.

M **O** In P*i*/Motif and P*i*/OPEN LOOK, we recommend protecting optionmenu widgets from data entry. If the widget is not protected from data entry, the user may type into the widget, but no text cursor appears in the widget. The lack of a text cursor may confuse users.

The ***selectscreen*** contains the values for the optionmenu. The value fields on the ***selectscreen*** must have the menu edit. The ***selectscreen*** is never actually displayed, but all menu fields on it appear as entries in the optionmenu. The values on the ***selectscreen*** may come from a database or other outside source. Since this screen is never displayed, two additional arguments, ***init*** and ***popup*** specify when **JAM** should open and close (but not display) the ***selectscreen***. Opening and closing the ***selectscreen*** initializes the `optionmenu` widget and performs any screen entry or exit processing on the ***selectscreen***. This allows the ***selectscreen*** populate the menu fields from a database call at screen entry.

The ***init*** argument may have the value `i` or `no_i`. A value of `i` indicates that the ***selectscreen*** should be opened and closed when the screen containing the `optionmenu` widget is initialized. A value of `no_i` indicates that it should not. ***init*** defaults to `i`.

The ***popup*** argument may have the value `p` or `no_p`. A value of `p` indicates that the ***selectscreen*** should be opened and closed when the pop–up is activated by the user. A value of `no_p` indicates that it should not. ***popup*** defaults to `no_p`.

Opening and closing the ***selectscreen*** may take a certain amount of time, particularly if a database query is involved. Therefore, you will probably wish to open and close the ***selectscreen*** as few times as possible. The default behavior, `(i, no_p)`, is appropriate if the values on the ***selectscreen*** do not change while the parent screen is displayed or if several fields on the screen use the same ***selectscreen***. Other combinations are appropriate in other circumstances.

NOTE: A combination of `(no_i,no_p)` is invalid, and causes the `optionmenu` pop–up to come up blank. The ***selectscreen*** ***must*** be opened and closed at least once, either upon initialization or pop–up.

Unless there is initial data in the **JAM** field, optionmenus with a ***selectscreen*** do not contain any value until the user posts the pop–up.

If you wish to pass a value from an optionmenu on one screen to another screen via the LDB, use the ***selectscreen*** flavor of optionmenu. The cycle field flavor of optionmenu cannot effectively pass a value. It simply passes the first occurrence of the array.

To convert a **JAM** field with an item selection screen to an `optionmenu`, specify the item selection screen as the ***selectscreen***. The user may then pop–up the optionmenu to make a selection or press the HELP key and open the item selection screen and make a selection that way.

WARNING: Do not attempt to post error messages from the field entry function of an `optionmenu` widget. If the field entry function causes a message dialog box to appear, the list of options closes immediately, before the user has a chance to make a selection.

Figure 56 illustrates optionmenus in Windows and Motif.

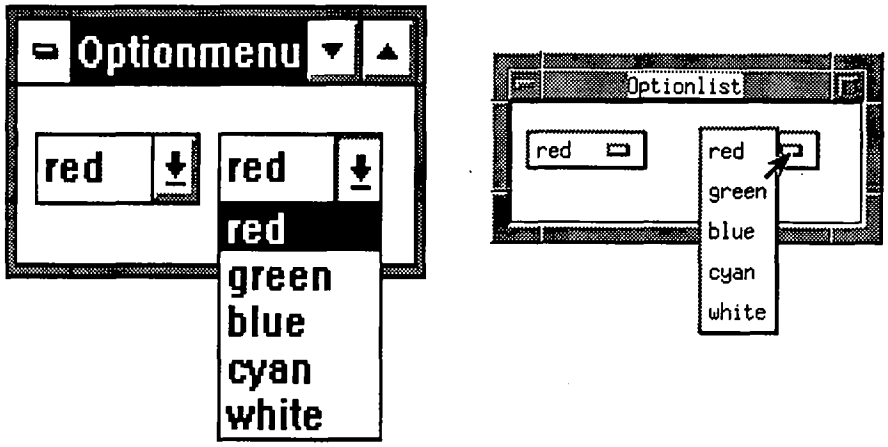


Figure 56: Option menus in Windows (left) and Motif (right). The right hand widget on each screen has its pop–up posted.

RELATED EXTENSIONS

```
# <<list>>
```

<<pixmap>>

associate a bitmap or pixmap with a label

SYNOPSIS

```
# <<pixmap(name)>>
```

TYPE

Field Extension

DESCRIPTION

Normally, a label displays a text string. This extension replaces that text string with the bitmapped image specified in ***name***. It may be used wherever a label widget appears. Specifically, in a protected field, or the label on a push button or toggle button. If you plan to use a bitmap on a push button, remember to place some text in the menu field; a blank menu field does not act as a menu.

Bitmaps display by default at the size they were created. If the field containing the bitmap has a height or width extension, this is respected.

In Pi/Windows, bitmaps are scaled to fit in the height and width specified.

In Pi/Motif and Pi/OPEN LOOK, bitmaps are truncated if they don't fit in the height and width specified.

Bitmap creation is GUI dependent.

In Windows, use the image editor or paintbrush utility to create bitmaps.

In Motif and OPEN LOOK, use the `bitmap` utility provided with X to create a bitmap, or create a pixmap file in the standard pixmap format, either as a text file or via a utility provided with your GUI.

Most distributions of X windows provide sample bitmaps as well.

In Pi/Windows, only a protected field can have a `pixmap`. Push buttons and toggle buttons *cannot*.

All bitmaps used in a **JAM** application must be installed in the Windows resource file for the application. For the **JAM** authoring tool, this file is called `wjxform.rc`. The syntax in the resource file is:

name `BITMAP` ***filename***

where ***name*** is the name of the bitmap and ***filename*** identifies the disk file containing the bitmap. Be sure to compile the resource file and link it with the application after making any changes. Refer to your MS Windows SDK documentation for more information on resource files.

If the bitmap file specified in the `pixmap` extension is not found, the extension is ignored.

In Motif, you can have a different bitmap for an armed versus unarmed push button or a selected versus unselected toggle button. The `pixmap` extension specifies the unarmed or unselected state of the button. The resources `armPixmap` and `selectPixmap` specify the pixmap in the armed or selected state. These resources may be specified for the entire class of push buttons or toggle buttons, for the buttons on a screen, or for individual named buttons. Chapter 7 discusses how to specify resources in Motif.

Under Motif, **JAM/Pi** searches first for a bitmap named ***name***, then for a bitmap named ***name.xbm***, then for a pixmap named ***name***, and finally for a pixmap named ***name.xpm***. The search path used depends on certain environment variables being set.

If XBMLANGPATH is set, **JAM/Pi** searches the path listed there for bitmaps.

If XBMLANGPATH is not set, but XAPPLRESDIR is set, then **JAM/Pi** searches directories in the following path:

```
.
$XAPPLRESDIR/bitmaps/application_class
$XAPPLRESDIR/bitmaps
$HOME
/usr/lib/X11/bitmaps/application_class
/usr/lib/X11/bitmaps
/usr/include/X11/bitmaps
```

If neither XBMLANGPATH nor XAPPLRESDIR is set, **JAM/Pi** searches directories in the following path:

```
.
$HOME/bitmaps/application_class
$HOME/bitmaps
$HOME
/usr/lib/X11/bitmaps/application_class
/usr/lib/X11/bitmaps
/usr/include/X11/bitmaps
```

In both cases, the language specific bitmap directories are also searched, depending upon the value of the LANG variable (eg.– $HOME/$LANG/bitmaps etc.).

If a bitmap is not found, **JAM/Pi** searches for a pixmap. The paths are the same as above, except that XPMLANGPATH replaces XBMLANGPATH and the word pix*maps* replaces the word *bitmaps* (eg.– $HOME/pixmaps).

If neither a bitmap nor a pixmap is found, the default bitmap is used.

O Under OPEN LOOK, **JAM/P*i*** searches first for a bitmap named ***name***, then for a bitmap named ***name.xbm***, then for a pixmap named ***name***, and finally for a pixmap named ***name.xpm***. The search path used depends on certain environment variables being set.

If XBMLANGPATH is set, **JAM/P*i*** searches the path listed there for bitmaps.

If XBMLANGPATH is not set, but XAPPLRESDIR is set, then **JAM/P*i*** searches directories in the following path:

```
.
$XAPPLRESDIR/bitmaps/application_class
$XAPPLRESDIR/bitmaps
$HOME
$OPENWINHOME/include/Xol/bitmaps
$OPENWINHOME/include/X11/bitmaps
/usr/lib/X11/bitmaps/application_class
/usr/lib/X11/bitmaps
/usr/include/X11/bitmaps
```

If neither XBMLANGPATH nor XAPPLRESDIR is set, **JAM/P*i*** searches directories in the following path:

```
.
$HOME/bitmaps/application_class
$HOME/bitmaps
$HOME
$OPENWINHOME/include/Xol/bitmaps
$OPENWINHOME/include/X11/bitmaps
/usr/lib/X11/bitmaps/application_class
/usr/lib/X11/bitmaps
/usr/include/X11/bitmaps
```

In both cases, the language specific bitmap directories are also searched, depending upon the value of the LANG environment variable, for example:

```
$XAPPLRESDIR/$LANG/bitmaps
```

If a bitmap is not found, **JAM/P*i*** searches for a pixmap. The paths are the same as above, except that XPMLANGPATH replaces XBMLANGPATH and the word pix*maps* replaces the word *bitmaps* (eg.– $HOME/pixmaps).

If neither a bitmap nor a pixmap is found, the default bitmap is used.

RELATED EXTENSIONS

```
# <<icon(name)>>
```

<<pointer>>

specify the pointer shape

SYNOPSIS

```
# <<pointer(shape)>>
```

TYPE

Screen Extension

DESCRIPTION

This screen extension specifies the shape of the mouse pointer on this screen. Some pointer shapes are listed below:

num_glyphs	dot	ll_angle	sb_v_double_arrow
X_cursor	dotbox	lr_angle	shuttle
arrow	double_arrow	man	sizing
based_arrow_down	draft_large	middlebutton	spider
based_arrow_up	draft_small	mouse	spraycan
boat	draped_box	pencil	star
bogosity	exchange	pirate	target
bottom_left_corner	fleur	plus	tcross
bottom_right_corner	gobbler	question_arrow	top_left_arrow
bottom_side	gumby	right_ptr	top_left_corner
bottom_tee	hand1	right_side	top_right_corner
box_spiral	hand2	right_tee	top_side
center_ptr	heart	rightbutton	top_tee
circle	icon	rtl_logo	trek
clock	iron_cross	sailboat	ul_angle
coffee_mug	left_ptr	sb_down_arrow	umbrella
cross	left_side	sb_h_double_arrow	ur_angle
cross_reverse	left_tee	sb_left_arrow	watch
crosshair	leftbutton	sb_right_arrow	xterm
diamond_cross		sb_up_arrow	

Strip off the `XC_` prefix when specifying the ***shape*** argument. The pointer shape may also be controlled with the `pointerShape` resource. The `pointerForeground` and `pointerBackground` resources control its color.

M In P*i*/Motif, pointer shapes are listed in the file `/usr/include/X11/cursorfont.h`.

O In P*i*/OPEN LOOK, pointer shapes are listed in the file `/$OPENWINHOME/include/X11/cursorfont.h`.

W In P*i*/Windows, this extension is not supported. Only the default cursor and busy cursor are available.

<<pushbutton>>

create a pushbutton widget

SYNOPSIS

```
# <<pushbutton>>
```

TYPE

Field Extension

DESCRIPTION

This extension creates a pushbutton widget from a field. Menu fields default to this widget type. For proper functionality a field with this extension should be a menu field, and it should be protected from data entry and tabbing. If it is not, the developer must add callbacks to handle selection processing. This is not recommended.

A push button widget performs an action when activated by the mouse or keyboard. It appears on the display as a button with a centered label and a drop shadow that causes it to protrude from the screen. Push buttons may be navigated via the keyboard or mouse just like character **JAM** menus.

You may wish to protect push buttons from clearing, as you would not want the user to inadvertently clear the label text in the button.

W **JAM** color settings and extended colors have no effect on push buttons. The color of push buttons in Windows is set in the Windows control panel, and there is one color scheme for all buttons throughout Windows. This scheme consists of a button face color, a button text color, and a button shadow color.

You can however use the `multiline` extension to create a push button whose color can be changed. See page 111.

M The distributed resource file in Pi/Motif contains a resource setting:

```
XJam*menuitem*alignment:  ALIGNMENT_BEGINNING
```

This setting changes the alignment of label text in push buttons to left justified, instead of the Motif default, center justified. You may change the value for this resource to `ALIGNMENT_CENTER` for center justification, or `ALIGNMENT_END` for right justification. For more information on resources, refer to Chapter 7.

O The distributed resource file in P*i*/OPEN LOOK contains a resource setting:

```
OLJam*area.RectButton.labelJustify:    center
OLJam*area.OblongButton.labelJustify:  center
```

This setting changes the alignment of label text in push buttons to center justified, instead of the OPEN LOOK default, left justified. You may change the value for this resource to `left` for left justification. Right justification is not supported. For more information on resources, refer to Chapter 7.

Figure 57 illustrates push buttons in Windows and Motif.

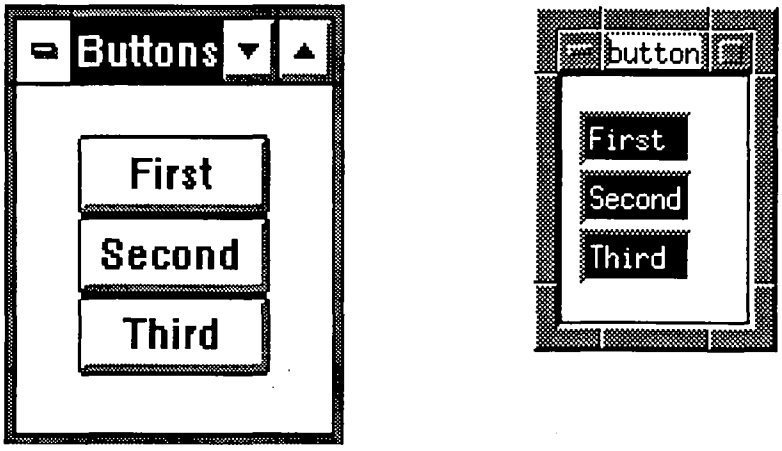


Figure 57: Push Buttons in Windows and Motif

<<radiobutton>>

create a radio style toggle button

SYNOPSIS

```
# <<radiobutton>>
```

TYPE

Field Extension

DESCRIPTION

This extension creates a radio style toggle button from a field. Members of radio button groups default to this widget type. To function properly the field must be a member of a group. If it is not, the developer must add callbacks to handle selection processing. This is not recommended.

M O In Pi/Motif and Pi/OPEN LOOK, only radio buttons with boxes become radio style toggle buttons. Radio buttons without boxes become in/out style toggle buttons. You can use this extension to create a radio style button from a radio button field without boxes. To avoid confusing the end-user, the `radiobutton` extension should be applied to each member of the radio button group.

One potential use for this extension is for a field that allows zero or one selection. In **JAM**, such a field must be created as a checklist group, since a radio button forces one and only one selection. The enforcement of only one selection in the checklist would be handled by the developer via a validation function. If the developer wished such a field to appear on the display as a radio style toggle button this extension would be necessary.

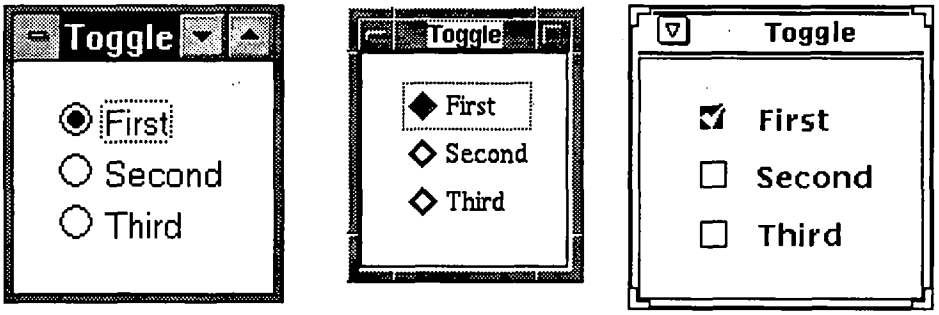


Figure 58: Radio buttons in Windows, Motif and OPEN LOOK.

<<scale>>

create a scale widget

SYNOPSIS

```
# <<scale(minimum-value, maximum-value, decimal-places)>>
```

TYPE

Field Extension

DESCRIPTION

This extension transforms a field into a scale widget. A scale is a combination widget consisting of a slider that runs between ***minimum-value*** and ***maximum-value***, and a label that changes to reflect the current value. ***decimal-places*** indicates the number of decimal places to be used in the value.

The contents of the underlying **JAM** field will be the value shown in the label, so you may use `sm_getfield` and `sm_putfield` to retrieve and set the value. The field should be long enough to hold the value and a sign, if necessary. A scale widget defaults to the size of the underlying **JAM** field. You may wish to give a scale a `width` field extension in order to widen it. The greater the range of values, the wider you should make the widget. You may also wish to give the field a no autotab edit.

For compatibility with character **JAM**, make a scale field digits only or numeric, and add a range check.

Figure 59 illustrates scale widgets in Windows and Motif.

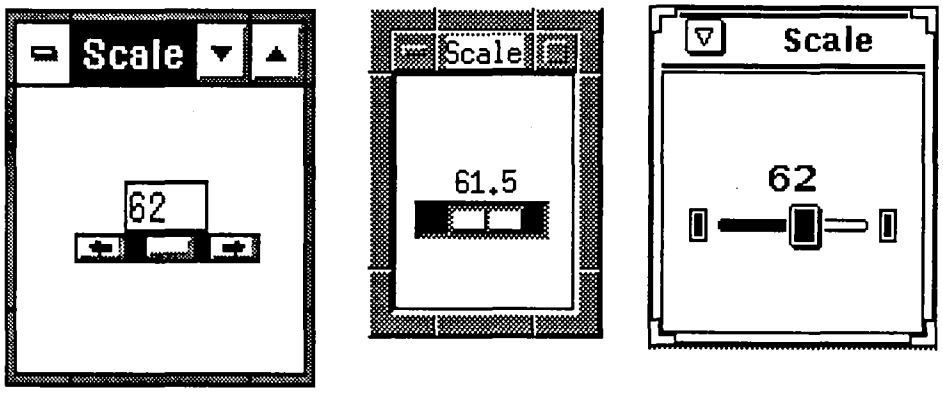


Figure 59: Scale widgets in Windows, Motif and OPEN LOOK.

<<space>>

equally space the elements of an array

SYNOPSIS

```
# <<space(distance [units])>>
```

TYPE

Field Extension

DESCRIPTION

Array elements are created by default as separate text widgets. These widgets are subject to the elastic grid. This means that there may not always be the same amount of space between array elements depending on how the grid has stretched. The `space` field extension guarantees equal spacing between each array element.

distance specifies the amount of space between each element. ***distance*** may be either an integer, in which case it represents the distance in pixels, or it may be any floating point number followed by a ***units*** suffix. ***units*** may be characters, grid units, inches, or millimeters. Refer to the chart on page 96 for an explanation.

The total height of an equally spaced vertical array is the sum of the heights of each element plus the space between the elements. The row height for the purposes of the elastic grid is the total height of the array divided by the number of rows it occupies. The same is true for the width and column size of a horizontal array.

The `space` field extension has no effect on multi–element arrays that are contained in single widgets, like those with the `multitext` or `list` extensions.

<<text>>

create a text widget

SYNOPSIS

```
# <<text>>
```

TYPE

Field Extension

DESCRIPTION

This extension creates a text widget from a field. Unprotected data entry fields default to this widget type. Protected fields can become text widgets with this extension. Their behavior depends on the specific protections. For example, the cursor will not stop at a field protected from tabbing.

If you use this extension on a selection field (ie.—a group member or menu field), the selection event will occur, but the user may have no way to tell, because the widget has no armed or selected state. Such use is not recommended.

Text widgets for left justified fields anchor by default on their left. Text widgets for right justified fields anchor by default on their right. The `halign` extension can be used to change the default alignment. See Chapter 3 for details on the positioning and widget sizing algorithms used in **JAM/***Pi*.

Figure 60 illustrates text widgets in Windows and Motif.

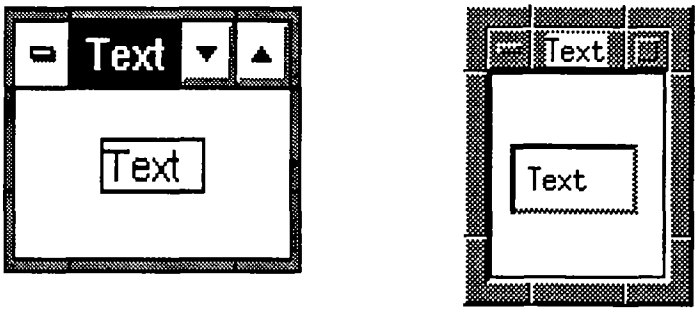


Figure 60: A Text widget in Windows and in Motif.

<<title>>

change the title bar on a screen

SYNOPSIS

```
# <<title(string)>>
```

TYPE

Screen Extension

DESCRIPTION

By default, each screen has a title bar. The contents of the title bar default to the name of the file that contains the screen binary, for example, `mainscrn.jam` (in P*i*/Motif, the extension is dropped in the title bar).

The `title` screen extension places ***string*** in the title bar of the screen, instead of the screen's file name. To blank out the text in the title bar, specify `title()`. To remove the title bar altogether, use the `notitle` extension.

In P*i*/Motif and P*i*/OPEN LOOK, title bars may also be set as a resource. The `title` screen extension overrides a title specified in the resource file.

RELATED EXTENSIONS

```
# <<notitle>>
```

<<togglebutton>>

create an in/out style toggle button

SYNOPSIS

```
# <<togglebutton>>
```

TYPE

Field Extension

DESCRIPTION

This extension creates an in/out style toggle button from a field. Members of radio button and checklist groups without boxes default to this widget type. To function properly the field must be a member of a group. If it is not, the developer must add callbacks to handle selection processing. This is not recommended.

M In Pi/Motif, you can use this extension to create an in/out style toggle button from a checklist or radio button field with boxes. To avoid confusing the end-user, the `togglebutton` extension should be applied to each member of the group.

Figure 61 shows a set of Motif in/out style toggle buttons.

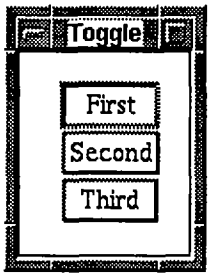
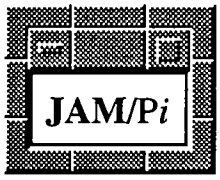


Figure 61: A set of Motif in/out style toggle buttons. The first item is selected.



Chapter 7
Setting Application Defaults

Each GUI provides its own method for setting defaults. P*i*/Motif uses *resource files*, and P*i*/Windows uses *initialization files*. Resource and initialization files are integral to the GUI. They control how the GUI and applications running under the GUI appear and act. The developer sets up the initial state of these files, but the user is free to change these settings. Allowing users to set their own preferences is fundamental to GUI philosophy.

7.1
RESOURCE AND INITIALIZATION FILES

The structure of resource and initialization files is determined by the GUI. Preferences are indicated by setting attribute/value pairs. **JAM/P*i*** applications use resource and initialization files to determine values for a variety of attributes including:

- Default fonts and colors
- Mapping between **JAM** colors and GUI colors
- GUI independent font and color names
- Application behavior

7.1.1
Resource and Initialization File Names

Each application may have an application specific resource or initialization file. The name of this application specific resource or initialization file is determined by the first argument to the **JAM** initialization routine, `sm_X11init`. This routine is called from the main routine of your application (usually either `jmain.c` or `jxmain.c`). If the

first argument to `sm_X11init` is, for example, the string "`myapp`", then the application specific resource file in P*i*/Motif and P*i*/OPEN LOOK is named `myapp`, and the application specific initialization file in P*i*/Windows is named `myapp.ini`. The default value for this argument in the distributed software is "`XJam`" in P*i*/Motif, "`OLJam`" in P*i*/OPEN LOOK and "`Jam`" in P*i*/Windows.

7.1.2

Structure of Resource and Initialization Files

Resource files and initialization files have a similar structure. Each is arranged as a list of attributes to be set along with a value for each attribute.

W Under Windows, initialization files take the form `attribute=value`, for example:

```
SystemFont=ANSI_VAR_FONT
```

The attribute being set in this case is `SystemFont` (the application default font). The value is understood to be any text to the right of the equal sign. Therefore the value is `ANSI_VAR_FONT`.

JAM initialization files are broken into sections that are set off by bracketed names. The sections are:

`[Jam Colors]`	A list of names and values for setting the sixteen **JAM** palette colors.
`[Jam Fonts]`	The application default font.
`[Jam Options]`	Behavior and appearance options.
`[Jam ColorTable]`	A list of GUI independent color names (aliases).
`[Jam FontTable]`	A list of GUI independent font names (aliases).

Comments in the initialization file are set off by a semicolon at the start of the line. The fragment below illustrates the structure of an initialization file. A sample initialization file appears on page 162.

```
[Jam Fonts]
;
;Enter the name of the application default font
;
SystemFont=OEM_FIXED_FONT
```

M Under Motif, resource files are arranged as colon separated attribute/value pairs, as in:

```
XJam*fontList:        fixed
```

The attribute being set in this case is `fontList` (the application default font). The value is understood to be any text to the right of the colon. White space directly after the colon is ignored. Therefore the value is `fixed`.

`XJam` is the class name. It restricts this resource to the `XJam` application. Resources may be further restricted to screens and even to individual widgets. The class name for a **JAM** application is determined by the first argument to the initialization routine `sm_Xlinit` (see above). The class name specified in `sm_Xlinit` may be overridden on the command line with the standard Xt command line argument `-name`.

Comments are indicated by starting the line with an exclamation point. Refer to your Motif documentation for a full explanation of resources and resource files.

O Under OPEN LOOK, resource files are arranged as colon separated attribute/value pairs, as in:

```
OLJam*font:        fixed
```

The attribute being set in this case is `font` (the application default font). The value is understood to be any text to the right of the colon. White space directly after the colon is ignored. Therefore the value is `fixed`.

`OLJam` is the class name. It restricts this resource to the `OLJam` application. Resources may be further restricted to screens and even to individual widgets. The class name for a **JAM** application is determined by the first argument to the initialization routine `sm_Xlinit` (see above). The class name specified in `sm_Xlinit` may be overridden on the command line with the standard Xt command line argument `-name`.

Comments are indicated by starting the line with an exclamation point. Refer to your OPEN LOOK documentation for a full explanation of resources and resource files.

7.1.3

Location of Resource and Initialization Files

In Windows, initialization files reside in the Windows directory.

In Motif, a resource database is constructed from several sources:

The application specific resource file, named by the class name of the application, is searched for in the directory: `/usr/lib/app-defaults` on the client machine. Resources specified here are global to all users of a particular application.

If the environment variable `XAPPLRESDIR` is set, the directory named in it on the client machine is searched for a resource file named by the application class name. This file may contain the user's or site administrator's preferences, and overrides settings in the application specific resource file.

Resources that are particular to one user's preference can be included in the `.Xdefaults` file in the user's home directory. The `.Xdefaults` takes precedence over other resource files. If you make changes to the `.Xdefaults` file while MWM is running, you must call `xrdb -load .Xdefaults` to reload the resource file.

Finally, command line options override any resources set in a resource file.

In OPEN LOOK, a resource database is constructed from several sources:

The application specific resource file, named by the class name of the application, is searched for in the directory: `/$OPENWINHOME/lib/app-defaults` on the client machine. These resources are global to all users of a particular application.

If the environment variable `XAPPLRESDIR` is set, the directory named in it on the client machine is searched for a resource file named by the application class name. This file may contain the user's or site administrator's preferences, and overrides settings in the application specific resource file. If `XAPPLRESDIR` is not set, a file with the application class name is looked for in `$HOME`, and if found, it is used.

Resources that are particular to one user's preference can be included in the `.Xdefaults` file in the user's home directory. The `.Xdefaults` takes precedence over other resource files. If you make changes to the `.Xdefaults` file while OPEN LOOK is running, call `xrdb -load .Xdefaults` to reload the resource file.

Finally, command line options override any resources set in a resource file.

7.2
COLORS

JAM/P*i* offers access to many more color choices than character **JAM**. Resource and initialization files provide a mapping between **JAM** colors and GUI colors. **JAM/P*i*** also provides a way to set up a GUI independent color naming scheme in the resource and initialization files. These colors can be used in the field and screen extensions.

7.2.1
Setting JAM Palette Colors

Character **JAM** provides sixteen colors to choose from, eight highlighted and eight unhighlighted. In the resource or initialization file, you can map these sixteen **JAM** colors to any of the colors supported by the GUI. This mapping between **JAM** colors and GUI colors defines your **JAM/P*i*** palette. Keep in mind that since end users have access to resource and initialization files, they are free to change the palette. The sixteen **JAM** colors that may be defined in the palette are:

`black`	`red`	`hi_black`	`hi_red`
`blue`	`magenta`	`hi_blue`	`hi_magenta`
`green`	`yellow`	`hi_green`	`hi_yellow`
`cyan`	`white`	`hi_cyan`	`hi_white`

In P*i*/Windows, palette colors are mapped to GUI colors in the `[Jam Colors]` section of the initialization file as follows:

jamcolor = ***color***

where ***jamcolor*** is a **JAM** color listed above and ***color*** is either,

an RGB value of the form: ***red/green/blue*** where ***red***, ***green*** and ***blue*** are numbers between 0 and 255.

a GUI independent color alias. Aliases are discussed in section 7.4.

For example,

```
Blue=0/0/255
Cyan=JYACC blue
```

In the above example, `JYACC Blue` is a color alias. You may wish to use the Windows palette feature on the Windows Control Panel to interactively mix your colors, and then note the values and transfer them to the initialization file.

W Note that there is a limitation in Windows for colors used as foregrounds. Foreground colors must be "primary" colors, ie—no dithered patterns. If you specify a non–primary color, Windows will round it up to a primary color if it is used as a foreground. Most monitors support sixteen primary colors, but some support more. These sixteen primary colors are mapped to the **JAM** palette colors in the `jam.ini` file, which is the initialization file distributed with **JAM**/*Pi*.

M **O** In *Pi*/Motif and *Pi*/OPEN LOOK, palette colors are mapped to GUI colors in the resource file. In Motif the syntax is:

```
XJam.jamcolor:      color
```

In OPEN LOOK the syntax is:

```
OLJam.jamcolor:     color
```

The variable ***jamcolor*** is a **JAM** palette color (listed above) and ***color*** is either,

a color name that appears in the `rgb.txt` file on your system. A sample `rgb.txt` file appears on page 166 in this chapter.

a hexadecimal RGB value. Hex specifications must be preceded by a # symbol. Refer to your GUI's *User's Guide* for details.

a GUI independent color alias. Aliases are discussed in section 7.4.

For example, in Motif:

```
XJam.blue:    DarkSlateBlue
XJam.green:   #00a800
XJam.cyan:    JYACC blue
```

or in OPEN LOOK:

```
OLJam.blue:   DarkSlateBlue
OLJam.green:  #00a800
OLJam.cyan:   JYACC blue
```

In the above examples, `JYACC Blue` is a color alias, while DarkSlateBlue is a GUI color listed in the `rgb.txt` file.

7.2.2
Colors Beyond the JAM Palette

For most applications, sixteen colors are sufficient. It is stylistically undesirable to flood screens with a multitude of hues, as they tend to distract the user. If additional colors beyond the sixteen defined in the the palette are needed though, they may be specified in the field or screen extensions.

The `fg` and `bg` extensions allow the developer to specify foreground and background colors for screens and widgets. These extensions can use either GUI specific colors or GUI independent color aliases. `fg` and `bg` are explained in Chapters 5 and 6. GUI independent color aliases are explained in section 7.4 of this chapter.

Motif Color Resources

Motif provides resources for changing the color of widgets and classes of widgets. **JAM/***Pi* respects these settings and allows them to override any color settings made within **JAM**. For example, a foreground color setting for the class of text widgets:

```
XJam*XmText*foreground:        blue
```

overrides all other foreground color for text widgets in the `XJam` application. A setting like the following changes the text widget foreground only for screen empscreen:

```
XJam*empscreen*XmText*foreground:    blue
```

OPEN LOOK Color Resources

OPEN LOOK provides resources for changing the color of widgets and classes of widgets. **JAM/***Pi* respects these settings and allows them to override any color settings made within **JAM**. For example, a foreground color setting for the class of text widgets:

```
OLJam*StaticText*fontcolor:        blue
```

overrides all other foreground color for text widgets in the `OLJam` application. A setting like the following changes the text widget foreground only for screen empscreen:

```
OLJam*empscreen*StaticText*fontcolor:    blue
```

Motif and OPEN LOOK Application Background and Foreground Resources

Motif and OPEN LOOK provide application–wide background and foreground color resources. These may be set from the command line or the resource file. **JAM/P*i*** interprets these resources to override the character **JAM** default background and foreground colors. Therefore, the application–wide background color replaces any unhighlighted black backgrounds, and the application–wide foreground color replaces any unhighlighted white foregrounds.

In the Motif resource file, the format for these resources is:

```
XJam*background:    color
XJam*foreground:    color
```

In the OPEN LOOK resource file, the format for these resources is:

```
OLJam*background:   color
OLJam*foreground:   color
```

On the command line in both GUI's, the format is:

```
-bg color -fg color
```

color is either a GUI color from `rgb.txt`, or hex value preceded by a # symbol. GUI independent color aliases may not be used with these resources.

The background and foreground resources offer a convenient method for allowing end users to set their own color preferences, provided that the developer has specified `unhighlighted black` as the background and `unhighlighted white` as the foreground in the display attributes for fields and screens, and that the fields and screens don't have `bg` or `fg` extensions which change their color.

NOTE: Don't confuse the application–wide background and foreground resources specified on the command line as `-bg` and `-fg` with the similarly named `bg` and `fg` field and screen extensions.

7.3 FONTS

JAM/*Pi* uses the standard GUI conventions for specifying fonts by name. For portability, font names can be aliased. Each application has a default font specified in the resource or initialization file. In addition, fonts may be specified for individual fields and screens.

7.3.1 Where Fonts are Specified

There are several places to set fonts in **JAM/*Pi***. Each type of specification has its own scope.

The Application Default Font

The application default font is specified in the resource or initialization file. In the absence of any other font specification, the application default font will be the font used for the entire application.

In *Pi*/Windows, the application default font is set via the `SystemFont` parameter in the `[JAM Fonts]` section of the initialization file, for example:

```
[JAM Fonts]
;
SystemFont=ANSI_VAR_FONT
```

Currently supported choices for `SystemFont` are:

`SYSTEM_FONT`	a proportional font (Windows uses it in pull-down menus).
`SYSTEM_FIXED_FONT`	a fixed width font. This is the font used in Draw Mode to complement `SYSTEM_FONT`.
`OEM_FIXED_FONT`	the PC, MSDOS character set. Use this font if your converted screens make use of character **JAM** line drawing. This is a fixed width font.
`ANSI_FIXED_FONT`	Courier, fixed width.
`ANSI_VAR_FONT`	Helvetica, proportional.

Other Windows font specifications cannot be used with the `SystemFont` setting.

In Pi/Motif, the application default font is set via the `fontList` resource:

```
XJam.fontList:   fontname
```

It may be overridden on the command line via the `-fn` switch as in:

```
xjxform -fn fontname
```

fontname is a Motif-specific font specification. Font aliases may not be used either in the `fontList` resource, or the `-fn` switch.

In Pi/OPEN LOOK, the application default font is set via the `font` resource:

```
OLJam.font:   fontname
```

It may be overridden on the command line via the `-fn` switch as in:

```
oljxform -fn fontname
```

fontname is an OPEN LOOK-specific font specification. Font aliases may not be used either in the `font` resource, or the `-fn` switch.

The Default Screen Font

The default screen font is either the application default font or a font specified via the `font` screen extension. A `font` screen extension overrides the application default font for a particular screen. In the absence of any other specification, this font will be used by all display text and widgets on the screen. The `font` screen extension takes either a GUI specific font name or a GUI independent alias. See page 89 for more on the `font` screen extension. See section 7.3.2 for an explanation of font naming, and section 7.4 for an explanation of font aliasing.

A Widget's Font

A widget's font is either the default screen font or a font specified via the `font` field extension. A `font` field extension overrides the default screen font for a particular widget. The `font` field extension takes either a GUI specific font name or a GUI independent alias. See page 89 for more on the `font` field extension. See section 7.3.2 for an explanation of font naming, and section 7.4 for an explanation of font aliasing.

7.3.2

Naming Fonts

Each GUI has its own font naming convention. JAM/P*i* can use either a GUI specific font name or a GUI independent font alias. This section describes the Windows, and Motif and OPEN LOOK font naming conventions. Section 7.4 describes aliasing.

Windows font naming

P*i*/Windows uses the following font naming convention:

fontname-pointsize`[-bold][-italic][-underline]`

fontname and ***pointsize*** are required values. `bold`, `italic` and `underline` are optional. For example:

```
Tms Rmn-24-bold
```

means Times Roman 24 point bold. Use the MS Windows Control Panel to find out what fonts are installed on your system. An additional font not listed in the Control Panel is `terminal`. This font is the same as the `OEM_FIXED_FONT` that can be specified in the initialization file as an application default font.

If the specified font is not found, it is either synthesized or replaced by a closely matching font according to the MS Windows GDI font mapping scheme. This scheme assigns weighted values to the various properties of a font, and then selects a font that is close to the one specified. Character set is given the greatest weight, followed by pitch, family, and face, then comes height and width, followed by weight, slant, underline and strikeout characteristics. Refer to *Reference Volume 1* of the MS Windows SDK documentation for a full description of the GDI and the various font characteristics.

Motif and OPEN LOOK font naming[1]

In Motif and OPEN LOOK, the simplest way to find out what fonts are available on your system is to run the `xlsfonts` program provided with the GUI. There are two common ways of specifying font names. The first is a simple font name, like "courier"

1. This section on Motif and OPEN LOOK font names is adapted from *Logical Font Description Conventions, Version 1.3, MIT X Consortium Standard.*

or "fixed". The second is for font names that conform to the XLFD font specification. These may be identified by the prefix "–". XLFD fonts use the following naming convention:

-foundry-family-weight-slant-width-style-pixel size-point size-x resolution-y resolution-spacing-average width-charset registry-charset encoding

Abbreviated definitions for the above values appear below. See the *X Protocol Reference Manual* for detailed explanations.

foundry Identifies the company that designed the typeface.

family Identifies the font family, for example, `courier`. Spaces are allowed in family names.

weight Nominal blackness of the font. Examples are: medium, demi–bold, bold.

slant A code that indicates the slant of the font. Options are:
`R` roman, `I` italic, `O` oblique, `RI` reverse italic, `RO` reverse oblique, `OT` other.

width Nominal width of characters. Examples are: normal, condensed, narrow.

style General style description, such as: serif, sans serif, informal, decorated.

pixel size The body size of the font in pixels at a particular point size and y resolution.

point size Device independent point size. Expressed in deci–points, eg.—`120` means 12 point type.

x resolution Horizontal resolution of the font in pixels per inch (dpi).

y resolution Vertical resolution of the font in pixels per inch (dpi).

spacing A code that indicates the spacing of the font. Options are:
`P` proportional, `M` monospaced, `C` character cell.

average width Average width of the characters in the font in deci–pixels (1/10th pixels). For the default screen font, this value determines the grid size. For a text widget, it determines the width of the widget.

charset registry The registration authority that owns the font's character set encoding.

charset encoding The registered name that identifies the coded character set.

Case is ignored in the font name specification. Wildcards may be used for any of the values, but the more exact a specification is, the more likely that the correct font is selected.

Example Font Specifications

```
-adobe-helvetica-bold-r-normal--24-240-75-75-p-130-iso8859-1

*helvetica-bold-r-normal--24-240*

-*helvetica*24*
```

The `xfontsel` Program

There is a program in some implementations of Motif and OPEN LOOK called `xfontsel` that provides a convenient interface for selecting fonts. It consists of a series of pull–down menus for selecting the various attributes of a font. Use this program to specify a font, and then "select" and paste the font specification into **JAM/P*i***.

The `xfontsel` screen is shown below.

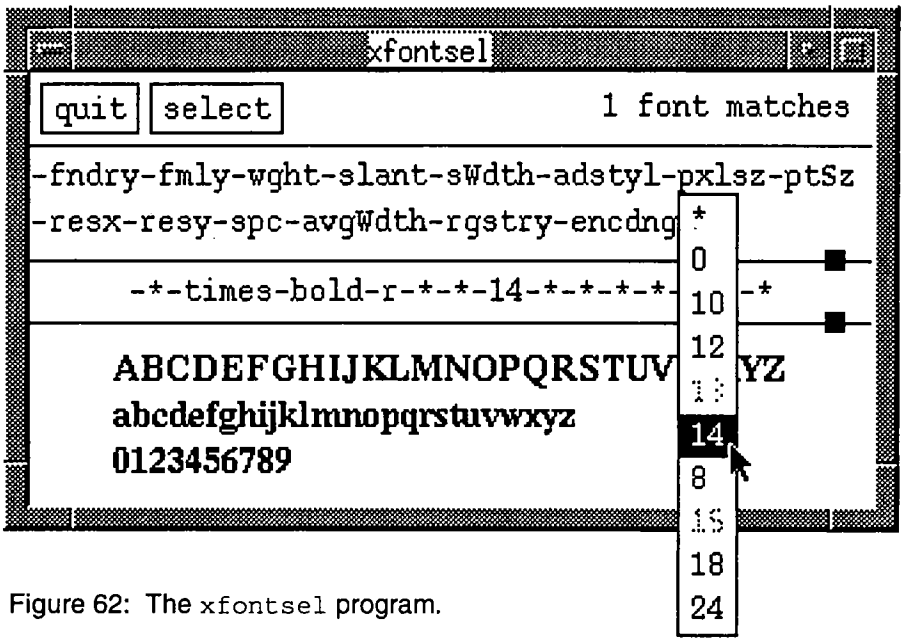


Figure 62: The `xfontsel` program.

To use `xfontsel`, select values from the pull–down menus associated with the various attributes to narrow down the list of fonts. A specification using the selected values appears in the center of the screen, and a sample of the first font that matches it appears beneath. The number of fonts that match the specification is listed at the upper right. On pull down menus, only those values that result in a valid font name based on the specification so far are available; all other values appear greyed out. If too many choices on menu appear greyed out, go to another selection and enter an asterisk.

When you are happy with the specification, click on the `select` button to copy it into the paste buffer. You may then paste this specification into the appropriate location in **JAM/P*i*** by clicking the middle mouse button or selecting paste from the menu bar.

7.4 ALIASING: GUI INDEPENDENT FONTS AND COLORS

Font and color aliasing allows the developer to specify GUI independent font and color names in the field and screen extensions. This enhances the portability of **JAM/P*i*** and simplifies the extensions, by moving the sometimes complex font and color specifications to the resource or initialization file, where they need be set only once.

Font and color aliases are made up by the developer, and their identities are resolved in an alias list in the resource or initialization file.

If you wish to use the **JAM** palette color names, like `hi_red`, in foreground or background extensions, you must add them to the list of color aliases.

In Windows, the alias list for fonts is contained in the `[JAM FontTable]` section of the initialization file, and the alias list for colors in the `[JAM ColorTable]` section. Each entry appears on its own line and consists of an alias followed by an equal sign and a Windows font or color specification. For example:

```
[JAM ColorTable]
JYACC Blue = 0/0/255
pumpkin = 255/128/14

[JAM FontTable]
JYACC Script = script-24-bold
italic = Tms Rmn-12-italic
```

Colors are specified as RGB values of the form:

red*/*green*/*blue

where ***red***, ***green*** and ***blue*** are numbers between 0 and 255.

Fonts are specified in the form:

***fontname*-*pointsize*`[-bold][-italic][-underline]`**

as described on page 155.

M **O** In P*i*/Motif and P*i*/OPEN LOOK, the alias list for fonts is contained in the `XJam*fonts` resource and the `OLJam*fonts` resource respectively. The alias list for colors is in the `XJam*colors` resource or the `OLJam*colors` resource. Each resource contains a newline separated list of alias pairs, made up of a GUI independent font or color name on the left, and a GUI dependent font or color on the right. For example, in Motif:

```
XJam*colors:  JYACC Blue    =   sky blue  \n\
              Champagne     =   #00eedd   \n\
              pumpkin       =   orange

XJam*fonts:   small   =   *-schumacher-*-6-*          \n\
              medium  =   *-helvetica-medium-r-*-10-*   \n\
              large   =   *-new century *-bold-i-*-20-*
```

Or in OPEN LOOK:

```
OLJam*colors: JYACC Blue    =   sky blue  \n\
              Champagne     =   #00eedd   \n\
              pumpkin       =   orange

OLJam*fonts:  small   =   *-schumacher-*-6-*          \n\
              medium  =   *-helvetica-medium-r-*-10-*   \n\
              large   =   *-new century *-bold-i-*-20-*
```

For each resource, every line except the last must end with a newline and a line continuation character. Leading and trailing whitespace is ignored.

GUI dependent colors are specified by name, or as hexadecimal RGB values. Color names are listed in the `rgb.txt` file and in your GUI user's manual. There is also sample list of colors on page 166.

Motif and OPEN LOOK fonts are specified via the XLFD font naming convention. See section 7.3.

Restrictions on Aliasing

Font and color aliases *may* be used in the field extensions, the screen extensions, and in the specification of the **JAM/P*i*** palette.

W In P*i*/Windows, a font alias *may not* be used for the `SystemFont` setting in the initialization file.

M O In Pi/Motif and Pi/OPEN LOOK, font and color aliases *may not* be used for the `foreground`, `background`, `font`, or `fontList` resources, nor may they be used on the command line with the `-fg`, `-bg`, or `-fn` arguments.

7.5 WINDOWS INITIALIZATION OPTIONS

The following sections describe options that are particular to Pi/Windows.

7.5.1 The `[Jam Options]` Section of the Initialization File

The following behavior and appearance options may be set in the `[Jam Options]` section of the application specific initialization file.

`GrayOutBackgroundForms`

This setting controls whether text on inactive screens is grayed out. While this behavior is usually desirable, there is a performance tradeoff associated with this functionality, since the background forms must be redrawn. `GrayOutBackgroundForms` defaults to `off`. To enable graying out, set this option to `on`.

`FrameTitle`

This setting controls the title text in the MDI frame around a **JAM** application. The default title string is the value of the first argument to `sm_X11init` in `jmain.c` or `jxmain.c` (see section 7.1.1).

`StartupSize`

This option, if set to `maximized`, brings up a **JAM** application in a maximized MDI frame. If it is set to `minimized`, then the application comes up in an iconified frame. Any other value brings up the application in a standard size frame. The default is to use a standard size frame.

StatusLineColor

This option sets the default background color for the **JAM** status line. For compatibility with other windows applications, it defaults to grey. Specify either an RGB value or a GUI independent color alias to change the default status line background. Messages with embedded display attributes can override the default background color.

SMTERM

This option overrides the SMTERM environment variable for P*i*/Windows applications. It allows both DOS and Windows to use **JAM** without the need to change the environment. To take advantage of this feature, set SMTERM to mswin in the initialization file, and to a DOS terminal type in the environment or SMVARS file. Example DOS terminal types are: cga, ega, mono, softcol and softbw.

7.5.2 The Windows Control Panel and win.ini File

Default attributes for Windows may be set from the "Windows Control Panel", usually found in the "Main" folder. From the Control Panel, you can setup the color scheme for Windows, as well as other defaults. The Control Panel alters the win.ini file, supplied by Microsoft. Refer to the MS Windows documentation for details of how to use the control panel

Some settings, such as the default color for buttons in Windows 1.2, can only be made by editing the win.ini file directly. A supporting document, the winini.txt file, is distributed with Windows. Read this file for instructions on altering win.ini.

7.5.3 Highlighted Background Colors in Windows

Note that in P*i*/Windows, highlighted background colors *are* different from unhighlighted background colors. In character **JAM** on a PC under DOS, there is normally no difference between highlighted and unhighlighted background colors.

7.5.4

Sample `jam.ini` File

```
[Jam Colors]
;
Black=0/0/128
Blue=JYACC Blue
Green=0/255/0
Cyan=0/255/255
Red=255/0/0
Magenta=255/0/255
Yellow=128/128/0
White=255/255/255
HBlack=0/0/0
HBlue=0/128/128
HGreen=0/128/0
HCyan=128/128/128
HRed=128/0/0
HMagenta=128/0/128
HYellow=255/255/0
HWhite=255/255/255

[Jam Fonts]
;
SystemFont=OEM_FIXED_FONT

[Jam Options]
;
GrayOutBackgroundForms=Off
;
FrameTitle=JAM
;
;StartupSize=Maximized
;
SMTERM=mswin
;
StatusLineColor=128/128/128

[Jam ColorTable]
;
JYACC Blue=0/0/128

[Jam FontTable]
;
Big Script=script-24-bold
```

7.6 MOTIF AND OPEN LOOK COMMON RESOURCE OPTIONS

This section describes resources that are common to P*i*/Motif and P*i*/OPEN LOOK.

7.6.1 Motif and OPEN LOOK Behavioral Resources

Three resources control the behavior of **JAM/P*i*** on an application-wide basis.

The `baseWindow` Resource

This resource controls whether a base window appears on the display. The base window is a special window that contains only a menu bar, a keyset, and a status line. If `baseWindow` is:

- true — (default) A base window appears on the display.
- false — No base window appears on the display. Any menu bar, keyset or status line that would have appeared in this window will be lost. See `formStatus` and `formMenus` to determine which status line and menu bars appear in the base window.

The `formStatus` Resource

This resource controls where status messages appear. Note that there is a difference between status and error messages. Error messages appear in dialog boxes in **JAM/P*i***. Status messages appear on the status line. This resource controls whether status messages appear on the base window's status line (the default), or on the active form's (or window's) status line. The existence of the base window is controlled by the `baseWindow` resource (see above).

There are five levels of status messages:

1. d_msg_line
2. wait
3. field
4. ready
5. background

Background status messages can only appear in the base window. If `formstatus` is:

- `false` (Default) All status messages appear in the base window. Individual screens have no status line of their own. If there is no base window (ie—`baseWindow: false`), then there is no status line at all.
- `true` Background status messages appear in the base window. All other status messages appear in a status line on the active screen. The status line on individual screens appears at the bottom of the screen. Only the active screen's status line is updated. If a screen is not active, then its status line is not updated.

The `formMenus` Resource

This resource controls whether individual forms (or windows) have their own menu bars. If `formMenus` is:

- false (Default) Only the base window displays a menu bar. Individual screens display no menu bar. Menu bars of all scopes, including screen–level, appear in the base window. If `baseWindow` is also false, then no menu bars appear at all.
- true Individual screens display their own menu bar. Screens display menu bars of the scope `KS_FORM` (screen–level) and `KS_OVERRIDE` (override–level). Only the active screen's menu bar is updated and active. Menu bars on inactive screens are inactive.

  The base window, if there is one, displays menu bars of the scope `KS_APPLIC` (application–level) and `KS_SYSTEM` (system–level). Whether the application–level or system–level menu bar is displayed in the base window may be toggled via the SFTS logical key. If there is no base window, then no system or application level menu bars are displayed.

Suggested Combinations of `baseWindow`, `formMenus` and `formStatus`

1. For compatibility with Pi/Windows and backward compatibility with controlled release versions of Pi/Motif, the default settings should be used:

```
XJam*baseWindow: true
XJam*formStatus: false
XJam*formMenus:  false
```

2. For full functionality with menu bars and status lines local to screens:

```
XJam*baseWindow: true
XJam*formStatus: true
XJam*formMenus:  true
```

3. If you wish to have no base window:

```
XJam*baseWindow: false
XJam*formStatus: true
XJam*formMenus:  true
```

Be sure *not* to use application level menu bars or background status messages with this third combination, as they will not appear.

For OPEN LOOK, replace the `XJam` in the samples with `OLJam`.

7.6.2

Restricted Resources

The following items in the distributed `XJam` file *must not* be changed:

```
XJam*...*translations
XJam*keyboardFocusPolicy
XJam*...*traversalOn
```

All other items (including: `Mwm*XJam*keyboardFocusPolicy`) may be changed at the developer's or user's discretion.

For OPEN LOOK, replace the `XJam` in the samples with `OLJam`.

7.6.3

Suggested Resource Settings

We strongly suggest the following resource setting.

```
XJam*focusAutoRaise:true
```

This setting will bring a **JAM** screen to the top of the display when it gets the focus. It is slightly different than the MWM resource of the same name.

For OPEN LOOK, replace the `XJam` in the samples with `OLJam`.

7.6.4

The `rgb.txt` File in Motif and OPEN LOOK

Motif and OPEN LOOK colors are listed in the `rgb.txt` file, often found in the directory `/usr/lib/X11` in Motif and in `$OPENWINHOME/lib` in OPEN LOOK. The `rgb.txt` file lists color names along with their red, green, and blue components. The colors appearing in this file are system dependent. Some common color names are:

alice blue
antique white
aquamarine
azure
beige
bisque
black
blanched almond
blue
blue violet
brown
burlywood
cadet blue
chartreuse
chocolate
coral
cornflower blue
cornsilk
cyan
dark goldenrod
dark green
dark khaki
dark olive green
dark orange
dark orchid
dark salmon
dark sea green
dark slate blue
dark slate gray
dark slate grey
dark turquoise
dark violet
deep pink
deep sky blue
dim gray
dim grey
dodger blue
firebrick
floral white
forest green
gainsboro
ghost white
gold
goldenrod
gray
green
green yellow
grey
honeydew
hot pink
indian red
ivory
khaki
lavender
lavender blush
lawn green
lemon chiffon
light blue
light coral
light cyan
light goldenrod
light gray
light grey
light pink
light salmon
light sea green
light sky blue
light slate blue
light slate gray
light slate grey
light steel blue
light yellow
lime green
linen
magenta
maroon
medium blue
medium orchid
medium purple
medium sea green
medium slate blue
medium turquoise
medium violet red
midnight blue
mint cream
misty rose
moccasin
navajo white
navy
navy blue
old lace
olive drab
orange
orange red
orchid
pale goldenrod
pale green
pale turquoise
pale violet red
papaya whip
peach puff
peru
pink
plum
powder blue
purple
red
rosy brown
royal blue
saddle brown
salmon
sandy brown
sea greenseashell
sienna
sky blue
slate blue
slate gray
slate grey
snow
spring green
steel blue
tan
thistle
tomato
turquoise
violet
violet red
wheat
white
white smoke
yellow
yellow green

7.7 MOTIF RESOURCE OPTIONS

The following sections describe resources and options that are particular to P*i*/Motif.

7.7.1 Motif Global Resource and Command Line Options

The resources in the table below are global settings for an application. They may also be specified on the command line, as may the standard X Toolkit command line options. Refer to the X Toolkit manual for a full list of command line switches.

NOTE: **D** indicates the default.

Resource	*Type*	*Command Line*	*Description*
`fontList`	string	`-fn` ***font***	Sets the application default font.
`foreground`	string	`-bg` ***color***	Sets unhighlighted white foregrounds to ***color.***
`background`	string	`-fg` ***color***	Sets unhighlighted black backgrounds to ***color.***
`setSensitive`	boolean	`-setSensitive` (on) `+setSensitive` (off) **D**	Controls whether screens that are not at the top of the window stack appear grayed out. You may wish to turn this off, since it slows down the application, and may cause other problems.
`ownColormap`	boolean	`-cmap` (on) `+cmap` (off) **D**	Tells **JAM** whether to use its own color map. Turning **JAM**'s color map on is useful only on systems with limited colors.
`cascadeBug`	boolean	`-cascadeBug` (on) `+cascadeBug` (off) **D**	Fixes a bug that appears in some versions of Motif 1.1. The bug causes popup menus to appear as small, empty boxes.

Resource	*Type*	*Command Line*	*Description*
`indicators`	boolean	`-ind` (on) `+ind` (off) **D**	Controls whether the Motif shift/scroll indicators are used. **NOTE:** There are also **JAM** shift/scroll indicators. To turn these off, use the `IND_OPTIONS` keyword in the Setup File. To change the characters used for the **JAM** indicators use the `ARROWS` keyword in the Video File. See the **JAM** *Configuration Guide* for more information.

The following illustrates a sample command line in P*i*/Motif:

```
xjxform -fn '-*courier*r*12' myscreen.jam
```

7.7.2

Widget Hierarchy in P*i*/Motif

Widgets are arranged in a parent–child hierarchy. The following tables describe the widget hierarchy in P*i*/Motif. This is useful to know if you wish to set resources for particular widgets or classes of widgets in an application. Refer to the OSF/Motif Programmer's Guide for more information on widgets, widget classes, and the resources associated with them.

Base Screen

The base screen in a **JAM** application is an `ApplicationShell` widget. Its class is given by the first argument to the `sm_X11init` initialization routine, and its name is the name of the application program (the value of `argv[0]` in `main`). If the `baseWindow` resource is set to `false`, then this shell is created but never displayed.

NOTE: Avoid application program names that contain periods or asterisks, as the resource parser interprets these as special characters. Screen name extensions, though, are removed when they are used as widget names.

By default, **JAM** has class name `XJam` and application name `xjxform`.

The widget hierarchy for the base Screen is:

Widget Class	*Name*
`ApplicationShell...` (class given by `sm_X11init`)	***application-name***
`XmMainWindow`	`main`
`XmDrawingArea`	`status`
`XmRowColumn`	`menubar`
`XmForm`	`workarea`
`XmPushButton`	`softkey`
⋮	⋮
`XmPushButton`	`softkey`

The `workarea` gets softkeys only when softkeys are enabled, and the `main` screen gets a menu bar only when menu bars are enabled (these are mutually exclusive). The `status` area is used for the **JAM** status line in the base screen.

Dialog Boxes

File selection dialog boxes are created by the `sm_filebox` library routine.

Message dialog boxes are created when a message needs to be posted. Error message dialogs are created by `XmCreateErrorDialog` and query message dialogs are created by `XmCreateQuestionDialog`. **JAM** specifies the message string, which buttons appear, and which button is the default. The **JAM** message call can specify the icon to appear. Other options, like the title bar text, can be set in the resource file.

The children of dialog boxes are handled by Motif. Refer to your Motif manual for details.

JAM Screens

The widgets used for **JAM** screens are all subclasses of the Motif `shell` widget. The `shell`'s parent is the `ApplicationShell`.

The widget hierarchy for **JAM** Screens is:

Widget Class	*Name*
`...TopLevelShell`	***screen-name***
`XmDialogShell`	`message_popup`
`XmMessageBox...`	`message`
`XmDialogShell`	`filebox_popup`
`XmFileSelectionBox...`	`fileBox`
`XmMainWindow`	`scroll`
`XmDrawingArea`	`clip`
`XmDrawingArea...`	`area`
`XmDrawingArea`	`status`
`XmScrollBar`	`scrollbar`
`XmScrollBar`	`scrollbar`
`XmRowColumn`	`menubar`

JAM screens have a status line only if the value of the `formStatus` resource is `true`. They have a menu bar only if `formMenus` is `true`.

New screens created in draw mode are named `shell` before they have been saved.

Since the name of the shell used for **JAM** screens is the screen name, resources may be restricted to a specific screen by beginning the specification with ***class* screen_name***. For example, `XJam*empscrn...` begins a specification for a screen named `empscrn` in an application of class `XJam`. Resources restricted to a named screen are equivalent to screen extensions. For example,

```
XJam*empscrn.background:        gold
```

is the same as specifying a `<<bg(gold)>>` as a screen extension on empscreen. The resource setting overrides the extension.

`area` is the parent widget for all the widgets on a **JAM** screen. If you place your own widgets on a **JAM** screen, you'll need the widget id of `area`. The library function `sm_drawingarea` returns the widget ID of `area`. A related function,

`sm_translatecoords`, translates **JAM** screen coordinates into pixel coordinates relative to the upper left hand corner of `area`.

Fields

JAM fields are created as child widgets of `area`. If a field has a name, its widget is given that name. If a field doesn't have a name, its widget is named `_fld#`, where # is the field number (this is analogous to the **JAM** `f2struct` utility). In a named array consisting of multiple widgets, each widget has the same name. Widgets that represent multiple fields take the name of their first field.

The library routine `sm_widget` returns the widget ID of a widget. Asterisks in the table below indicate which widget is returned by `sm_widget` in cases where there is more than one possibility. If the widget returned by `sm_widget` is not the one you are looking for, use `XtParent` to obtain the widget id of its parent. This is particularly useful when working with scale widgets and scrolling multiline and list box widgets.

Some entries in the table have prefixes or suffixes with their names. For example, ***field–name***`SW` indicates that the widget has name of the field followed by the literal characters `SW`.

The widget hierarchy for **JAM** fields is as follows:

Object	*Widget Class*	*Name*
Data Entry Field	`...XmText`	***field–name***
Data Entry Field with Indicators	`...XmDrawingArea`	***field–name***
	`XmText*`	***field–name***
	`XmArrowButton`	`indicator`
	⋮	⋮
	`XmArrowButton`	`indicator`
Protected Field	`...XmLabel`	***field–name***
Menu Field	`...XmPushButton`	***field–name***
Group Member	`...XmToggleButton`	***field–name***
Multiline Text	`...XmText`	***field–name***

Object	*Widget Class*	*Name*
Multiline Text with Scrollbars	...XmScrolledText	***field–name***SW
	XmText*	***field–name***
List Box	...XmList	***field–name***
List Box with Scroll Bars	...XmScrolledList	***field–name***SW
	XmList*	***field–name***
Optionmenu	...XmRowColumn*	***field–name***
	...XmMenuShell	popup_***field–name***_pane
	XmRowColumn	***field–name***_pane
	XmPushButton	***label–text***
	⋮	⋮
	XmPushButton	***label–text***
Scale	...XmScale	***field–name***
	XmScrollBar*	scale_scrollbar

To refer to a whole class of widgets, use the widget class. For example, `XJam*XmText` refers to all text widgets. To refer to a class of widgets on a screen, use the screen name followed by the widget class. For example, `XJam*empscreen*XmText` refers only to text widgets on the screen empscreen. To refer to an individual widget, use the screen name followed by the widget's name. For example, `XJam*empscreen*empname` refers only to the empname widget on the screen empscrn.

If the `indicators` resource is on (section 7.7.1), shifting and scrolling text widgets have indicator arrows. There can be up to four indicators, one for each direction.

In the optionmenu widget, the text field and the popup pane are linked through the `subMenuID` field of the `RowColumn` widget. Since the push buttons in the optionmenu are named by their contents, it is easier to set a resource for all the push buttons in an optionmenu than it is to set a resource for an individual button.

Display Text, Lines and Boxes

Display text, lines and boxes are child widgets of `area`. The hierarchy for display text and screen decoration widgets is as follows:

Object	*Widget Class*	*Name*
Display text	`...XmLabel`	`display`
Line	`...XmSeparator`	`line`
Box	`...XmFrame`	`box`
Frame	`...XmFrame`	`frame`

Menu Bars

Menu bars, submenus and pop–up menus are created within `RowColumn` widgets. Menu bars are children of either the base form's or an individual screen's `MainWindow`. Submenus are children of `MenuShells`, but the name of the shell is unclear, since Motif reuses these shells. If a new shell is created, its name will be `popup_`***submenu–name***. The best way to specify resources for a submenu is to use the form: `XJam*XmMenuShell.`***submenu–name***.

The hierarchy for menus and pop–up menus is as follows:

Object	*Widget Class*	*Name*
Menu Bar	`...XmRowColumn...`	***menu–name***
Submenu	`...XmMenuShell`	(name varies)
	`XmRowColumn...`	***submenu–name***
Pop–up Menu Bar	`ApplicationShell`	***application–name***
	`TransientShell`	`dummy`
	`XmMenuShell`	`popup_popupmenu`
	`XmRowColumn...`	`popupmenu`

Submenus pop up through the auspices of a `CascadeButton` widget. A submenu is tied to its `CascadeButton` via the `XmNsubMenuID` field of the button.

Items on menus and submenus are children of the menu's RowColumn widget. The hierarchy for items on menus and submenus is identical. It is as follows:

Menu Script Keyword	*Widget Class*	*Name*
separator	...XmSeparator	separator
title	...XmLabel	***label–text***
key or control (in top–level bar)	...XmCascadeButton	***label–text***
key or control (with indicator)	...XmToggleButton	***label–text***
key or control (without indicator)	...XmPushButton	***label–text***
menu	...XmCascadeButton...	***label–text***
edit	...XmPushButton...	***label–text***
windows	...XmPushButton...	***label–text***

The edit and windows submenus provide access to special **JAM** functions. Their contents are controlled by **JAM**, as opposed to being user designed with a menu script.

The hierarchy is shown below:

Object	*Widgets Class*	*Name*
Windows Menu	...XmRowColumn	windows
	XmPushButton	***window–name***
	⋮	⋮
	XmPushButton	***window–name***
	XmSeparator	sep1
	XmPushButton	windows_raise

Object	*Widgets Class*	*Name*
Edit Menu	...XmRowColumn	edit
	XmPushButton	edit_cut
	XmPushButton	edit_copy
	XmPushButton	edit_paste
	XmPushButton	edit_delete
	XmPushButton	edit_select

7.7.3

Sample Motif Resource File for JAM

```
!#############################################################
!###    Resource Specifications for XJam                   ###
!#############################################################

! Initial screen size:

XJam.geometry:                                  600x75+0+0

! Application-wide foreground and background:

!XJam*foreground:                               white
!XJam*background:                               dark slate gray

! Application default font:

!XJam*fontList:                                 fixed

! GUI focus policy:

XJam*keyboardFocusPolicy:                       explicit
XJam*focusAutoRaise:                            true
```

```
! GUI widget highlight and selection behavior:

XJam*highlightOnEnter:                  true
!XJam*highlightColor:                   dark orange
XJam*highlightThickness:                2
!XJam*allowOverlap:                     false

XJam*area.XmToggleButton.fillOnSelect:  true
XJam*area.XmPushButton.fillOnSelect:    true

! Label widget preferences:

XJam*area.XmLabel.marginWidth:          0
XJam*area.XmLabel.marginHeight:         0
XJam*area.XmLabel.highlightThickness:   0
XJam*area.XmLabel.highlightOnEnter:     false

! GUI indicator preferences:

XJam*indicator.width:                   15
XJam*indicator.height:                  15
XJam*indicator.highlightOnEnter:        false
XJam*indicator.shadowThickness:         0
XJam*indicator.traversalOn:             false
XJam*indicators:                        false

! Disable greying out of inactive screens:

XJam*setSensitive:                      false

! On some versions of Motif, a bug prevents the
! XmNcascadingCallback on a cascade button from
! being called, and therefore popup menus do not
! pop up. If this is so, set the following to true:

XJam*cascadeBug:                        false

! Under VMS, text widgets seem to grab the
! selection unless the following is set:

XJam*area*navigationType:               NONE
```

```
! Keyboard traversal activation:

XJam*area.XmPushButton.traversalOn:        true
XJam*area.XmToggleButton.traversalOn:      true
XJam*area.XmScale.traversalOn:             true
XJam*area*scale_scrollbar*traversalOn:     true
XJam*area.XmText.traversalOn:              true

! Label text alignment:

XJam*area.XmLabel.alignment:         ALIGNMENT_BEGINNING
XJam*area.XmToggleButton.alignment: ALIGNMENT_BEGINNING

! JAM palette colors:

XJam.black:        #000000
XJam.blue:         #0000a8
XJam.green:        #00a800
XJam.cyan:         #00a8a8
XJam.red:          #a80000
XJam.magenta:      #a800a8
XJam.yellow:       #a85400
XJam.white:        #a8a8a8
XJam.hi_black:     #545454
XJam.hi_blue:      #5454ff
XJam.hi_green:     #54ff54
XJam.hi_cyan:      #54ffff
XJam.hi_red:       #ff5454
XJam.hi_magenta:   #ff54ff
XJam.hi_yellow:    #ffff54
XJam.hi_white:     #ffffff

! Labels and keyboard mnemonics for the edit and windows
! menu bars:

XJam*XmMenuShell.windows.windows_raise.labelString:    Raise All
XJam*XmMenuShell.windows.windows_raise.mnemonic:  R
XJam*XmMenuShell.edit.edit_cut.labelString:        Cut
XJam*XmMenuShell.edit.edit_cut.mnemonic:           t
XJam*XmMenuShell.edit.edit_copy.labelString:       Copy
XJam*XmMenuShell.edit.edit_copy.mnemonic:          C
XJam*XmMenuShell.edit.edit_paste.labelString:      Paste
```

```
XJam*XmMenuShell.edit.edit_paste.mnemonic:        P
XJam*XmMenuShell.edit.edit_delete.labelString:    Delete
XJam*XmMenuShell.edit.edit_delete.mnemonic:       D
XJam*XmMenuShell.edit.edit_select.labelString:    Select All
XJam*XmMenuShell.edit.edit_select.mnemonic:       S

! Name of the RGB.TXT file to search for GUI color names:

XJam.rgbFileName: /usr/lib/X11/rgb.txt

! The standard JAM key file for X, "xwinkeys", maps
! unmodified, shifted, and control function keys 1-12
! into the JAM logical keys PF1-12, SPF1-12, and SFT1-12.
! This conforms to the standard key conventions used
! for JAM on character terminals.
!
! Unfortunately, these may conflict with the fallback or
! vendor-specific default bindings which Motif uses for
! its virtual keysyms. The following line disables all of
! the virtual keysyms within a JAM application.
! (Actually, the default binding for osfMenuBar is
! remapped to F25.  If we were to unmap it, the Motif
! library would reset it to F10.)
!
! If you prefer the standard Motif usage for the function
! keys, you can change the JAM key file to avoid the keys
! which conflict with Motif.  The following line can then
! be commented-out:

XJam*defaultVirtualBindings:        \n\
      osfMenuBar:       <Key>F25    \n\
      osfActivate:      <Key>KP_Enter \n\
      osfCancel:        <Key>Escape \n\
      osfDown:          <Key>Down   \n\
      osfLeft:          <Key>Left   \n\
      osfRight:         <Key>Right  \n\
      osfUp:            <Key>Up
```

```
! GUI independent color and font aliases for use in screen
! and field extensions:

XJam*colors: dark blue  = navy blue \n\
             champagne  = #00eedd   \n\
             pumpkin    = orange

XJam*fonts: small  = *-schumacher-*-6-*          \n\
            medium = *-helvetica-medium-r-*-10-* \n\
            large  = *-new century *-bold-i-*-20-*
```

7.8 OPEN LOOK RESOURCE OPTIONS

The following sections describe resources and options in P*i*/OPEN LOOK.

7.8.1 OPEN LOOK Global Resource and Command Line Options

The resources in the table below are global settings for an application. They may also be specified on the command line, as may the standard X Toolkit command line options. Refer to the X Toolkit manual for a full list of command line switches.

NOTE: **D** indicates the default.

Resource	*Type*	*Command Line*	*Description*
`font`	string	`-fn` ***font***	Sets the application default font.
`foreground`	string	`-bg` ***color***	Sets unhighlighted white foregrounds to ***color.***
`background`	string	`-fg` ***color***	Sets unhighlighted black backgrounds to ***color.***

Resource	*Type*	*Command Line*	*Description*
`setSensitive`	boolean	`-setSensitive` (on) `+setSensitive` (off) **D**	Controls whether screens that are not at the top of the window stack appear grayed out. You may wish to turn this off, since it slows down the application, and may cause other problems.
`ownColormap`	boolean	`-cmap` (on) `+cmap` (off) **D**	Tells **JAM** whether to use its own color map. Turning **JAM**'s color map on is useful only on systems with limited colors.

The following illustrates a sample command line in P*i*/OPEN LOOK:

```
oljxform -fn '-*courier*r*12' myscreen.jam
```

7.8.2

The OPEN LOOK `keepOnScreen` Resource

The `keepOnScreen` resource controls whether newly opened **JAM** screens are allowed to extend beyond the edge of the display. Normally, the OPEN LOOK window manager (olwm), allows this behavior. Setting this resource to true causes **JAM** to resize and move screens that the window manager initially places partially or totally off the display.

Once a screen has been opened, the user may move it off the edge of the display regardless of this resource setting.

7.8.3

Widget Hierarchy in P*i*/OPEN LOOK

Widgets are arranged in a parent–child hierarchy. The following tables describe the widget hierarchy in P*i*/OPEN LOOK. This is useful to know if you wish to set resources for particular widgets or classes of widgets in an application. Refer to the OPEN LOOK Programmer's Guide for more information on widgets, widget classes, and the resources associated with them.

Base Screen

The base screen in a **JAM** application is an `ApplicationShell` widget. Its class is given by the first argument to the `sm_X11init` initialization routine, and its name is the name of the application program (the value of `argv[0]` in `main`). If the `baseWindow` resource is set to `false`, then this shell is created but never displayed.

NOTE: Avoid application program names that contain periods or asterisks, as the resource parser interprets these as special characters. Screen name extensions, though, are removed when they are used as widget names.

By default, **JAM** has class name `OLJam` and application name `oljxform`.

The widget hierarchy for the base Screen is:

Widget Class	*Name*
`ApplicationShell...` (class given by `sm_X11init`)	***application-name***
`Form`	`main`
`Form`	`workarea`
`Control`	`softkeys`
`OblongButton`	`softkey`
⋮	⋮
`OblongButton`	`softkey`
`StaticText`	`status`
`Control`	`menubar`
`MenuButton`	`Edit`
`MenuButton`	`Windows`
`MenuButton`	***menu-name***
⋮	⋮
`MenuButton`	***menu-name***

JAM Screens

The widgets used for **JAM** screens are all subclasses of the OPEN LOOK `shell` widget. The `shell`'s parent is the `ApplicationShell`.

The widget hierarchy for **JAM** Screens is:

Widget Class	*Name*
`...TopLevelShell`	***screen–name***
`Form`	`scroll`
`StaticText`	`status`
`Control`	`menubar`
`MenuButton`	`Action`
`MenuButton`	***menu–name***
⋮	⋮
`MenuButton`	***menu–name***
`ScrolledWindow`	`clip`
`Scrollbar`	`Hscrollbar`
`Scrollbar`	`Vscrollbar`
`Bulletin`	`BulletinBoard`
`Bulletin`	`area`

JAM screens have a status line only if the value of the `formStatus` resource is `true`. They have a menu bar only if `formMenus` is `true`.

New screens created in draw mode are named `shell` before they have been saved.

Since the name of the shell used for **JAM** screens is the screen name, resources may be restricted to a specific screen by beginning the specification with ***class*screen_name***. For example, `OLJam*empscrn...` begins a specification for a screen named `empscrn` in an application of class `OLJam`. Resources restricted to a named screen are equivalent to screen extensions. For example,

```
OLJam*empscrn.background:      gold
```

is the same as specifying a <<bg(gold)>> as a screen extension on empscreen. The resource setting overrides the extension.

area is the parent widget for all the widgets on a **JAM** screen. If you place your own widgets on a **JAM** screen, you'll need the widget id of area. The library function sm_drawingarea returns the widget ID of area. A related function, sm_translatecoords, translates **JAM** screen coordinates into pixel coordinates relative to the upper left hand corner of area.

Dialog Boxes

Message dialog boxes are created when a message needs to be posted. Error and query message dialogs are created by XtCreatePopupShell with a widget type of noticeShell. **JAM** specifies the message string, which buttons appear, and which button is the default. Other options, like the title bar text, can be set in the resource file.

The children of dialog boxes are handled by OPEN LOOK. Refer to your OPEN LOOK manual for details.

Fields

JAM fields are created as child widgets of area. If a field has a name, its widget is given that name. If a field doesn't have a name, its widget is named _fld#, where # is the field number (this is analogous to the **JAM** f2struct utility). In a named array consisting of multiple widgets, each widget has the same name. Widgets that represent multiple fields take the name of their first field.

The library routine sm_widget returns the widget ID of a widget. Asterisks in the table below indicate which widget is returned by sm_widget in cases where there is more than one possibility. If the widget returned by sm_widget is not the one you are looking for, use XtParent to obtain the widget id of its parent. This is particularly useful when working with scale widgets and scrolling multiline and list box widgets.

Some entries in the table have prefixes or suffixes with their names. For example, ***field–name***SW indicates that the widget has the name of the field followed by the literal characters SW.

The widget hierarchy for **JAM** fields is as follows:

Object	*Widget Class*	*Name*
Data Entry Field	...TextField	***field–name***
Protected Field	...StaticText	***field–name***
Menu Field	...OblongButton	***field–name***
Checklist	...CheckBox	***field–name***
Radio Button	...RectButton	***field–name***
Multiline Text	...TextEdit	***field–name***
List Box	...ScrollingList	***field–name***
Optionmenu	...Control	***field–name***C
	StaticText*	***field–name***
	AbbrevMenuButton	***field–name***B
	...MenuShell	menu
	Form	menu_form
	Control	pane
	OblongButton	***label–text***
	⋮	⋮
	OblongButton	***label–text***
Scale	...Control	***field–name***C
	StaticText	***field–name***T
	Slider*	***field–name***

To refer to a whole class of widgets, use the widget class. For example, OLJam*TextField refers to all text widgets. To refer to a class of widgets on a screen, use the screen name followed by the widget class. For example, OL-Jam*empscreen*StaticText refers only to text widgets on the screen empscreen. To refer to an individual widget, use the screen name followed by the widget's

name. For example, `OLJam*empscreen*empname` refers only to the empname widget on the screen empscreen.

In the optionmenu widget, the text field and the popup pane are linked through the `subMenuID` field of the `RowColumn` widget. Since the push buttons in the optionmenu are named by their contents, it is easier to set a resource for all the push buttons in an optionmenu than it is to set a resource for an individual button.

Display Text, Lines and Boxes

Display text, lines and boxes are child widgets of `area`. The hierarchy for display text and screen decoration widgets is as follows:

Object	*Widget Class*	*Name*
Display text	`...StaticText`	`display`
Line	`...Stub`	`line`
Box	`...BulletinBoard`	`box`
Frame	`...BulletinBoard`	`frame`

Menu Bars

Menu bars, submenus and pop–up menus are created within `Control` widgets. Menu bars are children of either the base form's or an individual screen's `Form`. Submenus are children of `MenuShells`, but the name of the shell is unclear, since OPEN LOOK reuses these shells. If a new shell is created, its name will be menu. The best way to specify resources for a submenu is to use the form: `OLJam*MenuShell*`***button–name***.

The hierarchy for menus and pop–up menus is as follows:

Object	*Widget Class*	*Name*
Menu Bar	...Control...	menubar
Submenu	...MenuShell	menu
	Form	menu_form
	Control	pane
	OblongButton	***button–name***
	⋮	⋮
	OblongButton	***button–name***

Submenus pop up through the auspices of a MenuButton widget. A submenu is tied to its MenuButton via the XtNmenuPane resource of the button. This is the Control widget that the buttons are children of.

Items on menus and submenus are children of the menu's Control widget, except the title, which is a child of the menus form. The hierarchy for items on menus and submenus is identical. It is as follows:

Menu Script Keyword	*Widget Class*	*Name*
title	...Button	title
key or control (in top–level bar)	...MenuButton	***label–text***
key or control	...OblongButton	***label–text***
menu	...MenuButton...	***label–text***
edit	...OblongButton...	***label–text***
windows	...OblongButton...	***label–text***

The edit and windows submenus provide access to special **JAM** functions. Their contents are controlled by **JAM**, as opposed to being user designed with a menu script.

The hierarchy is shown below:

Object	*Widgets Class*	*Name*
Windows Menu	...MenuButton	windows
	...MenuShell	menu
	Form	menu-form
	Control	pane
	OblongButton	***window-name***
	⋮	⋮
	OblongButton	***window-name***
	Stub	sep1
	OblongButton	windows_raise
Edit Menu	...MenuButton	edit
	...MenuShell	menu
	Form	menu-form
	Control	pane
	OblongButton	edit_cut
	OblongButton	edit_copy
	OblongButton	edit_paste
	OblongButton	edit_delete
	OblongButton	edit_select

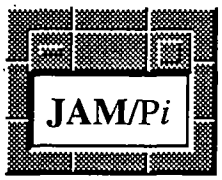
7.8.4

Sample OPEN LOOK Resource File for JAM

```
!#####################################################################
!###  Resource Specifications for OLJam                            ###
!#####################################################################

! Set the position with the geometry.
! Set the width of the Base Window by setting the width of the
! status line. Set the text alignment in the status bar with the
! gravity resource.
OLJam.geometry:                    +0+0
OLJam.main.status.width:           600
OLJam.main.status.recomputeSize:   false
OLJam.main.status.gravity:         west
OLJam*scroll.status.gravity:       west

! Set the look of the softkey area if they are used.
OLJam.main.workarea.softkeys.layoutType:       fixedcols
OLJam.main.workarea.softkeys.measure:          4
OLJam.main.workarea.softkeys.sameSize:         all

! Keep JAM screens completely on the display.
OLJam.keepOnScreen:                true

! Turning on/off of indicators are not supported in OLJam. They
! must be off.
OLJam*indicators:                  false

! Disable greying out of inactive screens.
OLJam*setSensitive:                false

! GUI focus policy.
OLJam*keyboardFocusPolicy:         explicit
!OLJam*allowOverlap:               false

! Set the positioning of text on windows and in buttons.
OLJam*area.StaticText.gravity:          west
OLJam*area.RectButton.labelJustify:     center
OLJam*area.OblongButton.labelJustify:   center
OLJam*area.CheckBox.labelJustify:       left
OLJam*area.CheckBox.position:           right
```

```
! Turn off Copy/Paste operations on scrolling lists.
OLJam*selectable:   false

! Set application-wide foreground and background
OLJam*foreground:   white
OLJam*background:   grey50

! Set color aliases.
OLJam*colors:       JAMfg = white /n/
                    JAMbg = grey50

! Set JAM palette colors
OLJam.black:        #000000
OLJam.blue:         #0000a8
OLJam.green:        #00a800
OLJam.cyan:         #00a8a8
OLJam.red:          #a80000
OLJam.magenta:      #a800a8
!OLJam.yellow:      #a85400
OLJam.yellow:       #e8e800
OLJam.white:        #a8a8a8
OLJam.hi_black:     #545454
OLJam.hi_blue:      #5454ff
OLJam.hi_green:     #54ff54
OLJam.hi_cyan:      #54ffff
OLJam.hi_red:       #ff5454
OLJam.hi_magenta:   #ff54ff
OLJam.hi_yellow:    #ffff54
OLJam.hi_white:     #ffffff

! Set application default font.
OLJam*font:  -*-lucida sans-bold-r-*-*-14-*

! Set font aliases.
OLJam*fonts: \n\
       small = -*-lucida sans-bold-r-*-*-12-*  \n\
       medium = -*-lucida sans-bold-r-*-*-18-* \n\
       large = -*-lucida sans-bold-r-*-*-24-*  \n\
       editorfont = -*-lucida sans typewriter-bold-r-*-*-18-*\n\
       JAMfont = -*-lucida sans typewriter-bold-r-*-*-18-*

! Set the labels for OK and Cancel buttons on Notices.
OLJam*NoticeShell*Control.okbutton.label:      OK
OLJam*NoticeShell*Control.cancelbutton.label:  Cancel
```

```
! Labels and keyboard mnemonics for the edit and windows menu bars
OLJam*MenuShell*windows_raise.label:            Raise All
OLJam*MenuShell*windows_raise.mnemonic:         R
OLJam*MenuShell*edit_cut.label:                 Cut
OLJam*MenuShell*edit_cut.mnemonic:              t
OLJam*MenuShell*edit_copy.label:                Copy
OLJam*MenuShell*edit_copy.mnemonic:             C
OLJam*MenuShell*edit_paste.label:               Paste
OLJam*MenuShell*edit_paste.mnemonic:            P
OLJam*MenuShell*edit_delete.label:              Delete
OLJam*MenuShell*edit_delete.mnemonic:           D
OLJam*MenuShell*edit_select.label:              Select All
OLJam*MenuShell*edit_select.mnemonic:           S

! Set no pointer warping when Notices are displayed to work around
a warping bug in olit patch T100451-39.
OLJam*NoticeShell.pointerWarping:               False

! Location of rgb.txt file to search for GUI color names.
OLJam.rgbFileName:          /usr/openwin/lib/rgb.txt


! The standard JAM key file for X, "xwinkeys", maps unmodified,
! shifted, and control function keys 1-12 into the JAM logical
! keys PF1-12, SPF1-12, and SFT1-12.  This conforms to the
! standard key conventions used for JAM on character terminals.
!
! Unfortunately, these may conflict with the fallback or vendor-
! specific default bindings which Motif uses for its virtual
! keysyms. The following line disables all of the virtual keysyms
! within a JAM application.  (Actually, the default binding for
! osfMenuBar is remapped to F25.  If we were to unmap it, the
! Motif library would reset it to F10.)
!
! If you prefer the standard Motif usage for the function keys,
! you can change the JAM key file to avoid the keys which conflict
! with Motif.  The following line can then be commented-out.

OLJam*defaultVirtualBindings:       \n\
       osfMenuBar:   <Key>F25       \n\
       osfActivate:  <Key>KP_Enter  \n\
       osfCancel:    <Key>Escape    \n\
       osfDown:      <Key>Down      \n\
       osfLeft:      <Key>Left      \n\
       osfRight:     <Key>Right     \n\
       osfUp:        <Key>Up
```



Chapter 8
Menu Bars

This chapter describes how to create and implement menu bars in **JAM/P*i***. Manual pages describing the menu bar library routines, which allow you to create, display and alter menu bars dynamically at runtime, are located in Chapter 12.

8.1
INTRODUCTION

Menu bars provide a convenient, permanently displayed area from which the user can select functions. A menu bar appears as a horizontal bar containing one or more menu bar headings. The contents of a menu bar can be changed according to the context. A menu bar can have several levels of submenus, which appear as vertical menus.

Menu bars are created as ASCII scripts. The script describes the content of the menu bar, the actions associated with each choice on the menu bar, and the display attributes of the items. Display attributes include grayed out choices, keyboard mnemonics, separators, and checked items. The menu bar utility, `menu2bin`, converts ASCII menu scripts into a binary format for inclusion in an application. `menu2bin` is described in section 12.2.

The content and selection of menu bars may be changed at runtime by library routines.

8.2
LOCATION OF MENU BARS

In P*i*/Windows there is only one menu bar per application. This menu bar appears at the top of the **JAM** frame, in accordance with the MS Windows Multiple Document Interface (MDI) specification. See section 4.1.2 for more on the MDI.

M O In Pi/Motif and Pi/OPEN LOOK, menu bars appear at the top of screens. There can be one menu bar per application, or menu bars for each screen. The `formMenus` resource controls this behavior.

If you choose to have one menu bar in your application (`formMenus: false`), the menu bar appears on the base screen (if there is one). The base screen is a special screen created by JAM/Pi that contains only a menu bar and a status line. The `baseWindow` resource controls the existence of the base screen.

If you choose to have multiple menu bars (`formMenus: true`), then each screen has its own menu bar in addition to the base screen's menu bar. Only the active screen's menu bar and the base screen's menu bar are active at any given time. The scope of a menu bar determines whether it appears locally on a screen or on the base screen.

For more on the `formMenus` and `baseWindow` resources, refer to Chapter 7.

8.2.1
Pop–Up Menu Bar in Motif and OPEN LOOK

M O In Pi/Motif and Pi/OPEN LOOK, the menu bar that appears on the base screen may also be accessed as pop–up menu bar via the right mouse button. The pop–up menu bar appears at the current mouse cursor position.

M **NOTE:** Some versions of Motif 1.1 have a bug in their handling of popup menus in which the widget is not notified that it has been activated. This causes pop–up menus to appear as small, empty boxes. To work around this problem, specify `XJam*cascadeBug: true` in the resource file, or use `-cascadeBug` on the command–line.

8.3
MENU BAR SCOPE

Just as with keysets, each menu bar has a scope. The scope is specified when the menu bar is installed. There may be an *application–level* menu bar, a *screen–level* menu bar, an *override–level* menu bar, a *system–level* menu bar, and any number of *memory–resident* menu bars. The table below describes the various menu bar scopes, and where they appear.

Scope	*Description*	*Location in Motif/OPEN LOOK*
KS_APPLIC	Application–level menu bar.	Base screen or pop–up.
KS_FORM	Screen–level menu bar.	Local to form if formMenus is true; otherwise, base screen.
KS_OVERRIDE	Override–level menu bar for help screens, zoom windows etc. Not used for error messages.	Local to form if formMenus is true; otherwise, base screen.
KS_MEMRES	Scope for storing memory–resident menus that can be accessed by menu bars at other scopes. Menus at this scope are stacked.	Not displayed.
KS_SYSTEM	System–level menu bar in the authoring utility jxform. A developer does not normally install a menu bar at this scope.	Base screen or pop–up.

If a window without a screen–level menu bar opens, the previously active menu bar remains displayed. This may be the screen–level menu bar from the previous screen, or the application–level menu bar, if the previous screen had no screen–level menu bar. If a form without a screen–level menu bar opens, then the application menu bar is active.

M **O** In Pi/Motif and Pi/OPEN LOOK, if formMenus is true, the screen–level menu bar appears local to the screen and the application–level menu bar appears on the base screen, so they may both be active simultaneously. If a screen without a screen–level menu bar opens, then no menu bar appears local to the screen.

When an override–level menu bar opens, the currently active menu bar is saved in a special stack (o_stack). When the override–level menu bar closes, this saved menu bar is restored. This stack may be 10 deep.

M **O** In Pi/Motif and Pi/OPEN LOOK, menu bars may appear on individual forms or on the base screen, depending on their scope and the value of the formMenus resource. Screen–level and override–level menu bars can appear either local to the form or on the base screen. Application–level and system–level menu bars are restricted to the base screen, but they may also be accessed as pop–up menus by pressing the third mouse button. If there is no base screen, then the menu bar that would appear on it is not displayed, although it can still be accessed as a pop–up.

8.4 THE MENU SCRIPT

Menu bars are created as ASCII scripts and converted to binary with the `menu2bin` utility. A menu script may contain specifications for one menu and one or more submenus. The first menu specification in a script file is the top level (horizontal) menu bar; subsequent menu definitions are for submenus.

8.4.1 Menu Script Structure

The general structure for specifying a menu is as follows:

```
menuname [global display options]
{
    "Label" action [modifiers] [display options]
    .
    .
#   comments
}
```

An alternative structure references an external menu, which is a menu that is already open or one that is stacked at the scope `KS_MEMRES`. This structure is as follows:

```
menuname external
```

The `external` keyword allows the developer to build menu bars in a modular fashion and reuse parts of menu scripts. Open menus are searched first for an external menu, then the `KS_MEMRES` stack is searched in a last opened, first searched order.

8.4.2 Menu Script Components

The various components of the general menu script structure are described below.

- ***menuname***

  identifies the menu. Any ***display options*** specified directly after the ***menuname*** are "global options" that apply to all relevant items in the menu. See ***display options*** below for an explanation. The curly braces are literal; they enclose the body of the menu.

- ***" label "*** is the text that appears in the menu entry. The label must appear in quotes. The menu bar compiler accepts labels up to 255 characters long, but in practice a menu bar displays only as many characters as will appear in the viewport. Backslash escapes may be used within the label for tabs, newlines and quotes if they are supported in your environment.

  An ampersand (&) is used as the keyboard mnemonic indicator in a label. Place the ampersand before the character in the label to be typed to select the entry from the keyboard. This character appears underlined in the menu entry. For example,

  ```
  E&xit
  ```

  produces the entry

  ```
  Exit
  ```

  where x is the keyboard mnemonic.

- ***action*** specifies the type of menu entry this is. Available keywords are:

  `title` specifies that ***label*** is the title of this menu. No ***modifier*** is allowed. The title must be the first entry in the menu.

In Pi/Windows, the `title` keyword is ignored.

  `menu` specifies that ***modifier*** is the ***menuname*** of a submenu.

  `key` specifies that ***modifier*** is a key to return when the entry is selected. Selecting the menu choice is equivalent to pressing the key. ***modifier*** can be a **JAM** logical key or a hex, binary or octal number. Specify hex with a leading `0x`. Specify binary with a leading `0b`. Specify octal with a leading `0`.

  `control` specifies that ***modifier*** is a **JAM** control string. Colon expansion is supported for menu bar control strings.

  `separator` produces a blank line. ***label*** is ignored. A separator can take a special separator ***display option***. Separators have no effect in horizontal menus.

  `edit` specifies that the edit submenu should appear. No ***modifier*** is allowed. The edit submenu contains the options: `Cut`, `Copy`, `Paste`, `Delete`, and `Select All`. These are useful for manipulating text in widgets.

`windows` specifies that the windows submenu should appear. No ***modifier*** is allowed. The windows submenu lists the ten topmost open screens by name. Selecting a screen from the list raises it to the top of the display. If the selected screen is a sibling of the screen at the top of the window stack, it becomes the top **JAM** screen.

In P*i*/Windows, the windows submenu (usually labelled "Window") also contains the entries: `Cascade`, `Tile` and `Arrange Icons`. These arrange screens and icons within the frame.

In P*i*/Motif and P*i*/OPEN LOOK, the windows submenu also contains a `raise all` option that raises all **JAM** screens to the top of the display, and layers them according to the window stack.

- Text ***display options***

  specify how an entry should appear. The display options for text entries are listed in the table below. Certain options are restricted to certain actions. A display option that is inappropriate for an action produces an error. More than one display option may be selected for an entry.

Display Option	*Actions*	*Description*
`inactive`	`menu`, `key`, `control`, `edit`, `windows`	Makes the entry inactive. The user may still click on the entry, but the entry has no effect.
`grayed` `greyed`	all actions	Grays out the entry's label and makes the entry non–selectable.
`indicator`	`key`, `control`	Shift all menu items to the right to leave room on the left for an indicator.
`indicator_on`	`key`, `control`	Turns an indicator on for this item. The indicator, often a check mark, denotes the state of a menu entry that serves as a toggle switch. If the `indicator` option is not also specified, this option shifts the menu. Indicators are ignored on horizontal menu bars.

Display Option	*Actions*	*Description*
`showkey`	`key`	Shows the keytop label from the key file to the right of the entry's label. If there is no keytop in the key file, then the key mnemonic is shown.
`help`	`menu`, `key`, `control`, `edit`, `windows`	Shifts an entry to the extreme right on a horizontal menu bar. Only one item may appear on the right. If the `help` item is not the last item in the menu bar specification, the compiler rearranges the items so it appears last.

- Separator ***display options***

  specify how a separator should appear. If no display option is specified, the separator is a single line. Only *one* separator display option may be selected. Separator display option keywords are GUI dependent. They are shown in the table below.

Display Option	*Interface*	*Description*
`menubreak`	Pi/Windows	Start a new line in a horizontal menu, or a new column in a vertical menu.
`single`	Pi/Motif	Single line. This is the default.
`double`	Pi/Motif	Double line.
`noline`	Pi/Motif	Draw no line, just leave a space.
`single_dashed`	Pi/Motif	Single dashed line.
`double_dashed`	Pi/Motif	Double dashed line.
`etchedin`	Pi/Motif	Single line etched into display.
`etchedout`	Pi/Motif	Single line that protrudes from display.
`single`	Pi/OPEN LOOK	Adds a blank line.

- Global ***display options***

  are global to the menu. They are specified directly after the ***menuname***. Global options affect all applicable menu entries. For example, if the global options are `showkey` and `noline`, all separators in the menu default to `noline` and all keys and control strings in the menu have `showkey`. Submenus and titles *do not* have `showkey` however, since it is not applicable to them.

  You cannot turn off a global option for an entry, but you can override a global separator option by specifying a new option for a particular separator.

- Comments

  begin with the # sign. Comments may appear on a line of their own anywhere within the script.

Keywords for ***action*** and ***display option*** are *not* case sensitive. ***labels*** and ***modifiers*** *are* case sensitive. White space characters in a script (space, tab, CR) are ignored by the menu bar compiler except when they separate keywords, so each menu specification can be quite compact.

8.4.3
Sample Menu Script

The following is an example of a menu script. Scripts must be compiled with `menu2bin` before they can be used. Figure 63 illustrates the menu that the sample script produces after it has been compiled.

```
# The first menu definition becomes the top level menu bar.

Main
{
        "Edit"  edit
        "Form"  menu FormMenu
        "Text"  menu TextMenu
        "Help"  menu HelpMenu help
        "&Quit" key  0x103
}

FormMenu
{
        "Form"      title
        "&New"      key PF1
        "&Open"     control "^jm_filebox file /usr/home * File"
        "&Close"    key PF3   inactive
        "&Save"     key PF3
        "Save &As"  key PF4
        ""          separator etchedin
        "O&ther"    menu      OtherMenu
}

# White space is ignored.

OtherMenu grayed showkey { "Other" title "Other&1" key PF1
"Other&2" key PF2 "E&xit" KEY EXIT }

# Keywords are not case sensitive.

TextMenu
{
        "&Cut"   KEY PF1
        "C&opy"  key PF2
        "&Paste" Key PF3
        ""       sEpArAtOR double menubreak
        "&Undo"  Key SPF1
}

# An external menu is one that is defined elsewhere, either
# in an open menu or at the scope KS_MEMRES.

HelpMenu external
```

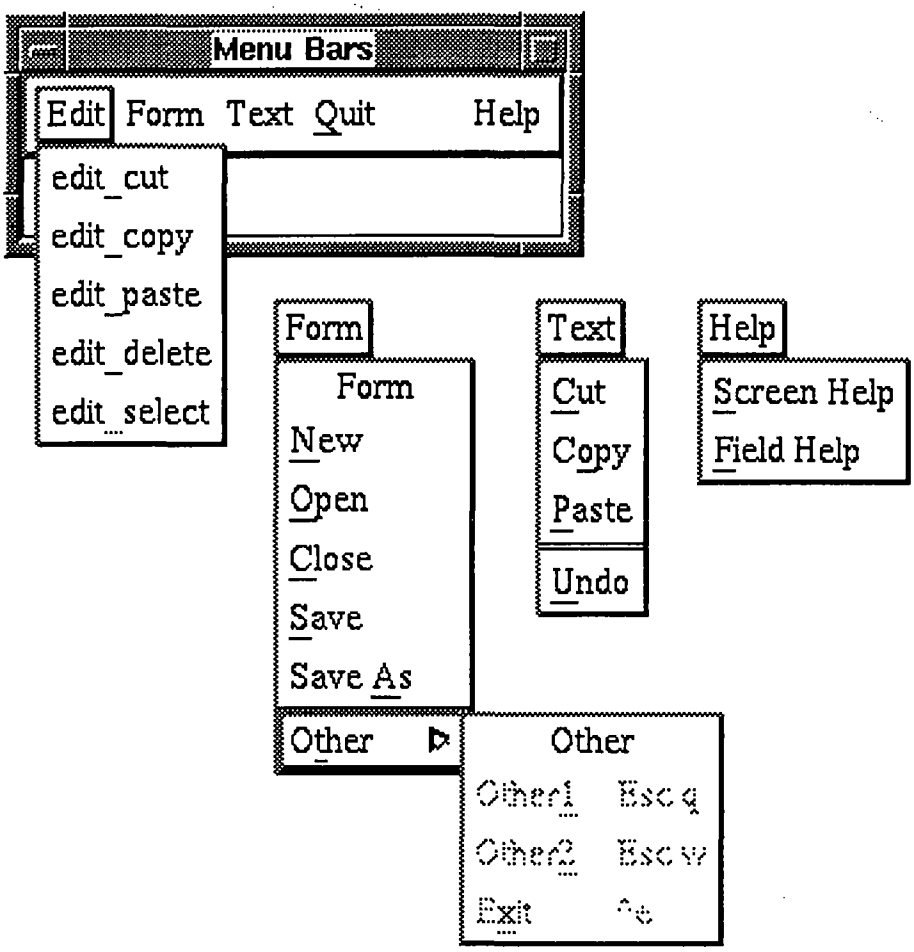


Figure 63: The menu bars produced by the sample menu script.

8.5 TESTING MENU BARS IN THE AUTHORING UTILITY

Menu bars can be tested in Application Mode of the authoring utility, but you must define a SFTS (soft key select) key in your keyboard translation file in order to do so. The

SFTS key toggles between user–defined menu bars and the system–level menu bar. In Application Mode, the default menu bar is the system–level menu bar. Use the SFTS key to toggle to your user–defined menu bars. Refer to the **JAM** *Utilities Guide* for details on using the `modkey` utility to edit a key translation file. Refer to the **JAM** *Configuration Guide* for an explanation of the key file.

8.6
MENU BAR LIBRARY ROUTINES

Library routines equivalent to those for keysets are provided to manipulate menus bars at runtime. Routines are available to create, display, close, and change the contents of menus bars. The table below summarizes these routines. For a detailed listing, see Chapter 12.

Routine	*Description*
`sm_c_menu`	close a menu bar
`sm_d_menu`	display a menu bar stored in memory
`sm_menuinit`	initialize menu bar support
`sm_mn_forms`	install menu bars in memory (in a custom executive)
`sm_mnadd*`	add an item to the end of a menu bar
`sm_mnchange*`	alter a menu bar item (eg– grey out an item)
`sm_mndelete`	delete a menu bar item
`sm_mnget*`	get menu bar item information
`sm_mninsert*`	insert a new menu bar item
`sm_mnitems`	get the number of items on a menu bar
`sm_mnnew`	create a new menu bar by name
`sm_r_menu`	read and display a menu bar from memory, a library or disk

NOTE: Library routines with an asterisk in the above table cannot be prototyped because they access an external data structure.

Prototyping Menu Bar Library Routines

You may wish to prototype the menu bar related library routines in order to use menu bars more flexibly. Prototyped library routines can be called directly from control strings and JPL procedures. Refer to the "Hook Functions" chapter in the **JAM** *Programmer's Guide* for an explanation of prototyped functions, and instructions on using and installing them. Refer to the *JPL Guide* for an explanation of how to use prototyped functions from JPL.

8.7 INSTALLING MENU BARS

Menu bars must be enabled and installed before they can be used in an application.

8.7.1 Enabling Menu Bars

In order to incorporate menu bars into your application, set `MENUS` to `1` in the appropriate `#define` in the `main` routine (`jmain.c` or `jxmain.c`). This causes the program to call the menu bar initialization routine, `sm_menuinit`. Alternatively, set the following flag in the makefile for your application: `-DMENUS`.

8.7.2 Installing Menu Bars of Various Scopes

The methods of installing menu bars depend on their scope.

Installing an Application–Level Menu Bar

Install an application–level menu bar with the library routine `sm_r_menu` or `sm_d_menu` using the scope `KS_APPLIC`. This is usually done in the main routine, `jmain.c` or `jxmain.c`, in the area reserved for code to be executed before the first screen is brought up.

Installing a Screen–Level Menu Bar

Menu bars are associated with screens in place of keysets; so to install a menu bar for a screen, insert the name of the menu bar file into the field for "Screen Level Keyset"

in the screen attributes window of the Screen Editor. A screen–level keyset may also be installed with the library routine `sm_r_menu` or `sm_d_menu` with a scope `KS_FORM`.

Installing Override–Level Menu Bars

Install an override–level menu bar with the `sm_r_menu` or `sm_d_menu` routine using the scope `KS_OVERRIDE`.

Installing Memory–Resident Menu Bars

Install memory–resident menu bars with the `sm_r_menu` or `sm_d_menu` routine using the scope `KS_MEMRES`. More than one menu bar can be loaded at this scope, and all are available simultaneously for use as an external menu by menu bars at other scopes. Installing a menu bar at this scope does not cause it to be displayed.

Installing the System–Level Menu Bar

The system–level menu bar is used only in the authoring utility. It is installed automatically by **JAM**.

8.7.3
Storing a Menu Bar in Memory

Binary menu bar files may be stored as disk files, as members of a library or in memory. A menu bar is stored in memory by converting it to a C structure with the `bin2c` utility, and then registering it to **JAM** with `sm_formlist`. For more information on this procedure, see the **JAM** *Programmer's Guide.*

NOTE: Do not confuse the memory–resident menu bar scope with the idea of storing menu bars in memory. The memory–resident menu bar scope, `KS_MEMRES` serves the purpose of keeping menu descriptions available for use as external menus. Storing a menu bar in memory means that it is compiled with your application, as opposed to being stored in a separate file.

8.8
USING MENU BARS EFFECTIVELY

Since menu bars are often the primary navigation tool in a GUI application, we suggest that you carefully consider which menu bar (or menu bars) appears in your application at any given point.

Use the sm_mnchange library routine to grey out or activate menu bar items in response to a change in context in the screen. Once you've altered a displayed menu bar, you must call sm_c_menu before calling sm_r_menu if you want to refresh the menu bar to its original state. This is because sm_r_menu does not reopen a menu bar if one with the same name is already open at a particular scope.

We suggest that you install a menu bar on each screen, rather than relying on the inheritance of menu bars from one screen to another. If you wish a screen to have no menu bar, install a dummy menu bar. If choose to rely on menu bar inheritance from screen to screen, be aware that altering an inherited screen–level menu bar changes the menu bar for the screen it was inherited from as well.

Instead of using the screen–level keyset field, you may wish to explicitly call sm_r_menu in the screen entry function and sm_c_menu in the screen exit function on each screen to open and close menu bars. This way you are always sure of which menu bar is displayed at any given time.

For greater efficiency, use the scope KS_MEMRES to store menus that are used by more than one menu bar.

8.9
MENU BARS VS. SOFT KEYS

Soft keys and menu bars are mutually exclusive, because they share the same programmatic hooks. The developer must choose whether to use one or the other. The selection of soft keys versus menu bars is made in the main routine, either jmain.c or jxmain.c, by initializing either soft key support or menu bar support. If an application is to use keysets in character **JAM** and menu bars in **JAM/Pi**, then the main routine should call the soft key initialization routine before it calls the menu bar initialization routine. The second library call will override the first in **JAM/Pi**, but will be ignored in character **JAM**.

8.9.1
Using Libraries to Store Menu Bars and Keysets

If an application uses menu bars on some platforms and soft keys on others, use libraries to store the keyset and menu bar files. Libraries provide a convenient method for switching between soft keys and menu bars on different platforms. If you name your keysets the same as your menu bars, but place the keysets in one library and the menu

bars in another, you may then specify which library to use on a particular platform with the `SMFLIBS` variable in the setup file. Use the `formlib` utility to create a library.

Refer to the **JAM** *Configuration Guide* for details on the setup file, and the **JAM** *Utilities Guide* for details on `formlib`.

8.9.2

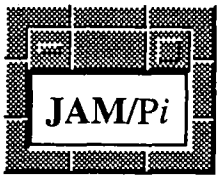
Converting Keysets into Menu Bars

Since soft keys and menu bars are mutually exclusive, the `kset2mnu` utility is provided to convert a keyset into an ASCII menu script.

Use the script output by this utility as a starting point for your menu bar. Since keysets are often organized differently than menu bars, you may wish to edit this script with a text editor before converting it to binary format. Menu bars usually have few direct actions listed on the top level menu; most headings are for submenus. Keysets, on the other hand, usually have direct actions in their first row, and then one or two additional rows of keys.

Menu bar are more versatile than keysets, so no direct conversion from keysets to menu bars is sufficient.

The `kset2mnu` utility is described in Chapter 12.



Chapter 9
Using the Mouse

9.1
Introduction

The mouse is generally the primary method for navigating through a GUI application. Mouse functionality in **JAM/P*i*** is similar to that in ***J**term* or **JAM** under DOS or OS2 character mode, although there are exceptions in cases where GUI dictated functionality differs from standard **JAM** functionality. In those cases, the GUI method is usually implemented.

9.1.1
Mouse Cursor Display

The mouse cursor is distinct from the **JAM** cursor. If a mouse is active, a mouse cursor will appear on the display.

W In P*i*/Windows, the mouse cursor appears as an I–bar when it is in a text field or display area. It appears as an arrow elsewhere.

The **JAM** cursor (or text caret) appears as a blinking block when the keyboard is in overwrite mode, and as a blinking vertical bar when the keyboard is in insert mode.

M In P*i*/Motif, the default mouse cursor is an arrow. Use the `pointer` screen extension to change its shape on a screen. The **JAM** cursor is a block in draw mode. In test and application modes, the **JAM** cursor is an I–bar in insert mode and a block in overstrike mode. A caret (secondary insertion cursor) may appear in one text widget as well, in the location where the mouse was last clicked. The caret has no function in **JAM**; it is merely a place holder created by the window manager.

In Pi/OPEN LOOK, the default mouse cursor is an arrow. Use the `pointer` screen extension to change its shape on a screen. The **JAM** cursor is a block in draw mode. In test and application modes, the **JAM** cursor is a carat in insert mode and a block in overstrike mode.

9.1.2
Mouse Buttons

The left mouse button positions the cursor, makes selections and operates widgets.

In Pi/Windows, **JAM** ignores any clicks or drags performed with the right and middle mouse buttons.

In Pi/Motif, the middle button, if available, is used for the paste operation in text widgets (see section 4.4). The right mouse button accesses the pop–up menu bar. The pop–up menu bar contains the same selections as the main menu bar, but avoids the inconvenience of moving the mouse cursor. See Chapter 8 for more on menu bars.

Note that if you only have a two button mouse, the GUI provides an equivalent, such as pressing both buttons, or pressing a key and button combination to replace the missing middle mouse button. Where the instructions below indicate to press the middle mouse button, simply use the equivalent instead.

In Pi/OPEN LOOK, the middle button, if available, can be used to extend a text selection. The right mouse button accesses the pop–up menu bar. The pop–up menu bar contains the same selections as the main menu bar, but avoids the inconvenience of moving the mouse cursor. See Chapter 8 for more on menu bars.

9.1.3
Mouse Functions

You may substitute a mouse click or drag for many keypresses, such as a `PF1`, `NL`, or the arrow keys. Below is a summary of how the mouse is used in **JAM**/Pi. For a complete description of editing features, or directions on creating fields, menus, groups, etc., please see the **JAM** *Author's Guide*.

Menu Bars

- To select a menu bar function, click on its menu bar heading to display its pull–down menu, and then click again on your selection; or press and hold the mouse button on its menu bar heading, and then drag the cursor down to your selection and release the mouse button.

  Menu bars may have several levels, called submenus. When the cursor is on a submenu heading, drag to the right to post the submenu. A submenu appears to the right of its heading in the parent menu.

  For detailed instructions on creating menu bars, refer to Chapter 8.

M
- In P*i*/Motif, on a pop–up menu bar, drag the cursor to your choice and then release the button to select it. If you drag the cursor to the heading for a submenu, and release the button without selecting an

M
- In P*i*/Motif, on a pop–up menu bar, drag the cursor to your choice and then release the button to select it. If you drag the cursor to the heading for a submenu, and release the button without selecting an

O
- In P*i*/OPEN LOOK, on a pop–up menu bar, drag the cursor to your choice and then release the button to select it, or click the right mouse button on your choice. If you click the right mouse button on a submenu, it comes up and remains posted. Click again to make a selection. The pop–up remains posted until a selection is made or

Focus

W
- In P*i*/Windows, the focus is always in the active screen, regardless of where the mouse pointer lies. A click is necessary to change the focus.

M
O
- In P*i*/Motif and P*i*/OPEN LOOK, a mouse click is sometimes necessary to change the focus, depending on the context, and the settings in the resource file. For details, refer to the section 4.1.3 on focus behavior.

- Click on a sibling of the active screen to change the focus to the sibling. If the click is on a field, then the **JAM** cursor moves to that field. If the click is on a display or protected area, then the **JAM** cursor is restored to the same location inside the sibling window that it was in when the window was last visited, or to the first unprotected field if the screen was never visited.

  NOTE: Windows that are not siblings of the active screen *cannot* be made active. A click within one of these stacked windows does not change **JAM**'s focus.

M
O
In P*i*/Motif and P*i*/OPEN LOOK, the previous discussion applies when explicit focus is set. If pointer focus is set, a click is not necessary. Simply move the mouse cursor into a sibling window to activate it. The **JAM** cursor returns to the location it was in when the window was last visited. See your Motif manual for a discussion of focus behavior.

- A sibling window may also be activated by selecting its name from the optional "Window" heading on the menu bar. The names of all open screens appear under this heading, but only those that are siblings of the active screen can be selected.
- A screen that cannot be activated may still be moved and resized by dragging on its border (see below).

- In Pi/Windows, when you move or resize a screen that cannot be made active, it rises to the top while the mouse button is depressed, but the active screen regains the top position when the button is released.

Move, Offset and Resize

- In **JAM/Pi**, the **JAM** viewport (VWPT) key is *not* available. **JAM**'s viewport functions are replaced by the GUI's screen manipulation protocols. These are described in detail in the *Microsoft Windows User's Guide* or the *X Window System User's Guide*, and briefly here as well. To manipulate screens, do the following:

  MOVE — Drag on the title bar of the screen.

  RESIZE — Move the mouse cursor to the border or corner of the screen. The cursor changes shape. Drag the corner or border to the desired size.

  When a viewport is smaller than its underlying screen, scroll bars appear.

  NOTE: Unlike character **JAM**, a viewport may be larger than its underlying screen. When the viewport is as large as or larger than the underlying screen, the scroll bars disappear.

  OFFSET — Drag the scroll bar at the bottom or right hand border of the screen, or click on a scrolling arrow.

  The move and resize functions can be suppressed with the `nomove` and `noresize` screen extensions.

Moving the Cursor and Making Selections

- In Draw Mode, clicking anywhere on a screen moves the **JAM** cursor to the mouse cursor's position.
- Clicking on a regular data entry field moves the **JAM** cursor to the field. The **JAM** cursor moves to the character position of the mouse

cursor within the field. If you click on display text or a tab–protected field in Test or Application Modes, **JAM** ignores the click.

- Clicking on a checklist item moves the **JAM** cursor to that item and either selects or deselects it, depending on its current state. If a checklist group has the autotab edit, the **JAM** cursor goes to the next item in the group when the user selects an item.
- Clicking on a radio button item moves the **JAM** cursor to that item and selects it. Radio button items may only be deselected by selecting another item. If a radio button group has the autotab edit, the **JAM** cursor automatically leaves the group when a selection is made.
- Clicking twice in a yes/no field toggles its value. The first click moves the cursor to the field, and the second click executes the toggle. The click is translated as if the opposite value was typed (i.e., via `unget-key`). If the field has an `autotab` edit, the second click toggles the value and then moves the **JAM** cursor to the next field.

  BEWARE: *Do not click twice when choosing to edit JPL text from the screen attributes window of the screen editor.*

  If there is already text in the JPL window, the toggle field contains a `y`. A double click toggles the value to to `n`, and the existing JPL text is permanently lost. Instead of double clicking, click once (or tab to the field) and press `y` on the keyboard.
- Clicking once on the "OK" or "Cancel" button in a dialog box acknowledges the message. Dialog boxes replace character **JAM** error and acknowledgement messages. Pressing the space bar (or other `ERR_ACK_KEY`) also clears these messages. See section 4.2.
- Dragging and releasing (or clicking once) on an onscreen application menu makes a selection. The selection is made on the "click up".
- When using soft keys, clicking on a key label is the same as pressing that key.
- Clicking on a status line keytop is the same as pressing the logical key.

Scrolling and Shifting

- Scroll or shift a field by dragging the cursor beyond its edge in the direction you wish to scroll or shift. Note that this method has the effect of selecting the text that you drag through, so be sure not to type a character while the text is highlighted, or the text will be deleted.

- Drag the scroll bar or click on the scroll arrows to shift or scroll widgets with scroll bars.

Editing Text

- When an area of text is selected, typing from the keyboard deletes the selected text. The first typed character replaces the text. In overstrike mode, as you continue to type, subsequent characters type over existing characters. In insert mode, subsequent characters are inserted.

- In Pi/Windows, you can cut or copy text in a text widget, and then paste it into another text widget (or onto the screen as display text in Draw Mode). Drag across text to select it. Choose Cut or Copy

- In Pi/Motif, you can cut, copy, and paste text in text widgets. Drag across text to highlight it. The highlighted text becomes the primary selection. Reposition the cursor by moving the mouse and then click

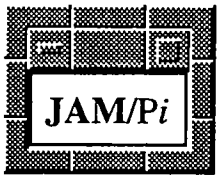
- In Pi/OPEN LOOK, you can cut, copy, and paste text in text widgets. Drag across text to highlight it, or click the extend button to select the range of text between the cursor and the mouse pointer.
- If more text is pasted than fits into a field, the overflow characters do not flow into the next field. Instead, overflow characters are truncated. Similarly, characters rejected by a character edit are thrown away. Use the cut or copy keys or menu bar choices to manipulate the selected text. To paste buffered text, reposition the cursor to the new location and choose paste from a key or menu bar.
- Multiple occurrences may be copied and pasted from one array to another. If you attempt to paste data into more occurrences than are available, the overflow is truncated.

Select Mode

- In select mode, click on a field or area of display text to select or deselect the text, depending on its current state. Selected items may be cut, copied, moved, or altered, using **JAM** select mode functionality.
- In select mode, click the mouse to mark the corners of a selection box. First click on the position where the box is to begin. Then choose the "box" option. Finally, click on the opposite corner of the box. All fields and display text inside the box are surrounded by selection brackets.
- When using the move or copy functions in select mode, either click once at the new position to move or copy the selection or use the cursor keys. The cursor keys are more exact in this case.

Miscellaneous

- Click on a character in the `Special characters` window to move the cursor to the character and select it. Note that not all characters are available in all fonts.



Chapter 10
GUI Specific Features

This chapter deals with issues that are specific to a particular GUI.

10.1
OVERSTRIKE MODE IN P*i*/MOTIF AND P*i*/OPEN LOOK

Normally Motif and OPEN LOOK do not support overstrike mode. P*i*/Motif and P*i*/OPEN LOOK *do* support overstrike mode in text widgets. In fact, overstrike mode is the default text entry mode in **JAM**/P*i*, just as it is in character **JAM**.

10.2
INTERFACING WITH THE GUI LIBRARY

JAM/P*i* provides three library routines that enable the developer to refer to **JAM** windows and screen objects as GUI objects. They provide an interface between **JAM**/P*i* and GUI–provided library functions.

The first routine, `sm_widget`, returns the widget id of (or handle to) a widget on a screen. The second routine, `sm_drawingarea`, returns the widget id of (or handle to) the GUI window that contains the current **JAM** screen. The widget id is necessary in order to manipulate a GUI object or refer to it from a GUI library function.

The third routine, `sm_translatecoords` converts **JAM** screen coordinates (line and column) into pixel coordinates relative to the upper left hand corner of the drawing area, which is the container widget used to hold a **JAM** screen. The pixel coordinates are required if you wish to place external objects on **JAM** screens.

`sm_widget`, `sm_drawingarea` and `sm_translatecoords` are fully documented in Chapter 12. Included on the manual page for `sm_translatecoords` is an example illustrating how to use these functions to place a bitmap on a **JAM** screen in P*i*/Windows.

A demonstration program that uses external graphics is provided in source form with **JAM/***Pi*. It is called `winpie` in P*i*/Windows, `xpie` in P*i*/Motif. This program also illustrates how to use `sm_drawingarea` and `sm_translatecoords`. Refer to this code, and your GUI toolkit documentation, for detailed information on how to proceed.

10.3
SYSTEM COMMANDS IN P*i*/WINDOWS

In P*i*/Windows, in order to view the output of a DOS system command, you must create a Program Information File (PIF) for the command, using the MS Windows PIF editor. The PIF editor is located in the Accessories program group. See the MS Windows User's Guide for details on the PIF editor.

Disable the "Close Window on Exit" option in your PIF, so the user may view the output of the DOS command once it has terminated. If this option is not disabled, the command will terminate and return to Windows before the user has had a chance to view the output.

If a command is likely to produce more than one screenful of output, create a batch file that pipes the command's output through a utility such as `more`. Then create a PIF file that calls the batch file.

Once a PIF has been created, call the PIF instead of the command. For example, if you created a `chkdsk.pif` that calls the DOS `chkdsk.com` command, you would type

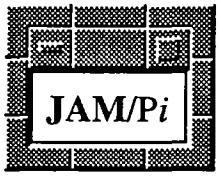
```
!chkdsk.pif
```

to call the command from a control string rather than

```
!chkdsk.com.
```

To call a DOS command contained in the `command.com` utility, use the `/c` option switch to `command.com` or write a batch file that calls the command directly. For example, to get a directory listing, create a PIF that calls `command.com` as the "Program Filename" and use `/c dir` as the "Optional Parameter" in the PIF editor. Alternatively, write the following batch file, and create a PIF for it instead:

```
REM View directory listing one page at a time.
dir|more
```



Chapter 11
Conversion Issues

This chapter deals with issues relevant to applications that are being converted from character **JAM** into **JAM/P*i***.

11.1
BACKGROUND HIGHLIGHTS

On certain terminals (such as the PC), there is normally no such thing as a highlighted background color, so setting the highlight attribute for a background has no effect. In **JAM/P*i*** though, highlighted background colors are supported, giving you much more flexibility in color selection. If you normally set the background highlight on, then when you convert your applications, be sure to check the color to make sure it is to your liking.

11.2
LINE DRAWING

Line drawings do not convert well into **JAM/P*i*** screens. Use the `hline`, `vline`, `box`, and `frame` extensions instead. See Chapters 5 and 6 for more information.

In Pi/Windows, if you select the base font to be `OEM_FIXED_FONT`, line drawings will look reasonable, unless the screen contains groups with checkboxes or other widgets that expand in size.

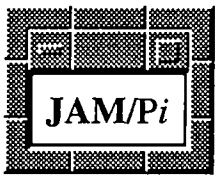
11.3
JAM VERSION 4 APPLICATIONS

JAM version 4 applications must first be converted into version 5 applications before being transferred to JAM/P*i*.

11.4
JAM VERSION 5 APPLICATIONS

Screens from JAM version 5.0 or later can be opened under JAM/P*i*.You will probably wish to embellish these screens with extended colors and fonts, and to reposition and resize some of the screen objects. As mentioned in previous chapters, display text should be converted into protected fields to take advantage of positioning and extended features.

If you have used menu fields and submenus to simulate pull down menus in character JAM, you will want to convert these into menu bars, and then eliminate the menu fields from the screen. Since menu bars are often the primary navigational tool in GUI applications, you may wish to take advantage of them.



Chapter 12
Library and Utility Reference

This chapter is divided into two sections, Library Routines and Utilities.

12.1
JAM/Pi LIBRARY ROUTINES

JAM/Pi library routines are available for each GUI interface as noted in the "Supported Interfaces" section on each man page. These routines are not portable to character **JAM**.

GUI Library Interface Routines

The following routines give the developer access to the widgets created by **JAM/Pi** so that they may interact with them directly.

`sm_drawingarea`	get the widget id (or handle) of the current **JAM** screen
`sm_translatecoords`	translate screen coordinates to display coordinates
`sm_widget`	get the widget id (or handle) of a particular widget

Menu Bar Routines

The menu bar routines are analogous to the equivalent keyset routines. Keysets are documented in the **JAM** *Author's Guide*, and the keyset routines are documented in the **JAM** *Programmer's Guide*. Menu bars are described in detail in Chapter 8.

You may wish to prototype some of these routines, in order to increase your flexibility in dealing with menu bars. Prototyping library routines allows them to be called directly from control strings and JPL procedures. Refer to the "Hook Function" chapter in the **JAM** *Programmer's Guide* for an explanation of prototyped functions, and instructions

on their installation and use. Refer to the *JPL Guide* for an explanation of how prototyped functions may be used from JPL.

The following routines create, alter, install and display menu bars.

`sm_c_menu`	close a menu bar
`sm_d_menu`	display a menu bar stored in memory
`sm_menuinit`	initialize menu bar support
`sm_mn_forms`	install menu bars in memory
`sm_mnadd`	add an item to the end of a menu bar
`sm_mnchange`	alter a menu bar item
`sm_mndelete`	delete a menu bar item
`sm_mnget`	get menu bar item information
`sm_mninsert`	insert a new menu bar item
`sm_mnitems`	get the number of items on a menu bar
`sm_mnnew`	create a new menu bar by name
`sm_r_menu`	read and display a menu bar from memory, a library or disk

File Selection Box Routines

The following routines initialize and open a file selection dialog box.

`sm_filebox`	open a file selection dialog box
`sm_filetypes`	set up a list of file types for a file selection dialog box

Miscellaneous Routines

`sm_adjust_area`	refresh the current screen
`sm_win_shrink`	trim the current screen

NOTE: The header file `smdefs.h` must be included to run any **JAM** library routine. Other header files required by specific routines are noted on each routine's manual page.

sm_adjust_area

refresh the current screen

SYNOPSIS

```
#include "smpi.h"

void sm_adjust_area()
```

DESCRIPTION

This routine redisplays the current screen, recalculting the positioning and sizing. It is useful if a widget has changed size, due to its protection changing, or the screen being toggled in or out of menu mode.

If a widget is changed to or from a label widget as a result of its protection being changed, it will most likely shrink or stretch. Similarly, fields that have the menu edit but are not protected from data entry will change their nature depending on whether the screen is in menu mode or data entry mode. This may change the size of their widgets. **JAM** does not automatically refresh the screen under these conditions, which may cause widgets to overlap. Use `sm_area_adjust` to refresh the screen and recalculate the relative positioning of objects.

SUPPORTED INTERFACES

Pi/Windows
Pi/Motif
Pi/OPEN LOOK

sm_c_menu

close a menu bar

SYNOPSIS

```
#include "smsoftk.h"

int sm_c_menu(scope)
int scope;
```

DESCRIPTION

This routine closes the menu bar at the given `scope`. It frees all memory associated with the menu bar. If the menu bar is currently displayed, it is removed at the next delayed write.

Scope	*Description*
KS_FORM	Screen–level menu bar.
KS_APPLIC	Application–level menu bar.
KS_OVERRIDE	Override–level menu bar.
KS_MEMRES	Memory–resident menu bar.
KS_SYSTEM	System–level menu bar.

When a menu bar with a `scope` of `KS_OVERRIDE` closes, the previously displayed menu bar is restored from the override stack

If `scope` is `KS_MEMRES`, the last menu bar opened at that scope is closed.

To refresh a menu bar with a new copy from disk (or memory), first call `sm_c_menu`, and then call `sm_r_menu` or `sm_d_menu`.

RETURNS

0 if there is no error.
–2 if there is no menu bar currently at `scope`.
–3 if menu bars are not supported or `scope` is out of range.

RELATED FUNCTIONS

```
sm_d_menu(menu, scope);
sm_r_menu(name, scope);
```

SUPPORTED INTERFACES

P*i*/Windows
P*i*/Motif
P*i*/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"

/* Close the current JAM window's menu. */

sm_c_menu( KS_FORM );
```

sm_d_menu

display a menu bar stored in memory

SYNOPSIS

```
#include "smsoftk.h"

int sm_d_menu(menu, scope)
char *menu;
int scope;
```

DESCRIPTION

The parameter menu is the address of a menu bar stored in memory. The utility bin2c is used to create program data structures from disk based menus. These structures are then compiled into your application and added to the memory–resident screen list, described in Chapter 9 of the **JAM** *Programmer's Guide.*

scope is one of the mnemonics listed in smsoftk.h and shown in the table below.

Scope	*Description*
KS_FORM	Screen–level menu bar.
KS_APPLIC	Application–level menu bar.
KS_OVERRIDE	Override–level menu bar.
KS_MEMRES	Memory–resident menu bar.
KS_SYSTEM	System–level menu bar.

If there is currently a menu bar with the specified scope, the name of that menu bar is compared with menu. If they are the same, the routine returns immediately. Thus to refresh a menu bar with a new copy from memory, call sm_c_menu first.

If scope is KS_OVERRIDE, the currently displayed menu bar is saved in a stack (o_stack). When the override menu bar closes, the saved menu bar is restored. This stacking is performed only for a scope of KS_OVERRIDE. This scope is used for help screens, zoom windows, etc. The stack is fixed at 10 deep.

If scope is KS_MEMRES, the menu bar is read from memory and added to the stack of memory–resident menu bars for use as external menus.

For all other scopes, the menu bar is read from memory and installed. The old menu bar at this scope, if any, is freed. If the menu bar at this scope is currently displayed, it must be refreshed. This fact is marked and the actual refresh is performed at the next delayed write.

RETURNS

0 if no error occurred during display of the menu bar.
–1 if the format is incorrect (ie, not a menu bar).
–3 if menu bars are not supported or the `scope` is out of range.
–5 if there is a malloc failure.

In the case of an error, the previously displayed menu bar remains displayed.

For all errors except –3 a message is posted to the operator.

RELATED FUNCTIONS

```
sm_c_menu(scope);
sm_r_menu(name, scope);
```

SUPPORTED INTERFACES

P*i*/Windows
P*i*/Motif
P*i*/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
  ...
extern char customer_menu[];
  ...

/* Display the customer menu as the application-level menu.
 * Customer_menu was created using bin2c.
 */

sm_d_menu( customer_menu, KS_APPLIC );
```

sm_drawingarea

get the widget id of the current **JAM** screen

SYNOPSIS

W

```
#include "mswin.h"
HWND sm_drawingarea();
```

M O

```
Widget sm_drawingarea();
```

DESCRIPTION

Provides the widget id of the current **JAM** screen. This function in conjunction with `sm_translatecoords` is useful when placing objects such as bitmapped graphics or custom widgets on a **JAM** screen. Refer to the source listing for the pie chart demonstration provided with **JAM/P*i*** for a detailed example of how to import graphics and use these functions. An example is also provided on the manual page for `sm_translatecoords`.

RETURNS

Returns NULL if there is no current screen.
Otherwise:

W A handle to the window.

M O The widget id as a Widget.

RELATED FUNCTIONS

`sm_translatecoords(column, line, column_ptr, line_ptr);`

`sm_widget();`

SUPPORTED INTERFACES

P*i*/Windows
P*i*/Motif
P*i*/OPEN LOOK

EXAMPLE

```
/* This is a Pi/Windows example */

#include "smdefs.h"
#include <windows.h>

int current_window_maximize( void )
{
  /* This is a JAM prototype-able function which maximizes the current
   * JAM window.  It is the equivalent of having the user click the
   * window's maximize button.  The function sm_drawingarea returns
   * the window handle for the currently active JAM window.
   */

  PostMessage( sm_drawingarea(), WM_SYSCOMMAND, SC_MAXIMIZE, 0 );
  return( 0 );
}
```

sm_filebox

open a file selection dialog box

SYNOPSIS

```
#include "smpi.h"

int sm_filebox(buffer, length, path, file_mask, title, flag)
char *buffer
int length
char *path
char *file_mask
char *title
int flag
```

Built–in control function variant:

```
^jm_filebox fieldname path file_mask title flag
```

DESCRIPTION

This function opens a file selection dialog box. A file selection box allows the user to browse through a directory tree and select a file by name. The implementation details of the dialog are GUI dependent, but the function's parameters are the same across GUI's.

`buffer` is used to contain the full pathname of the user's selection. `length` is the length of `buffer`. It is up to the developer to provide a buffer large enough to hold the pathname.

`path` is the initial path for the directory tree. `file_mask` is a filter for narrowing down the files in `path`. It should contain at least one wildcard character.

`title` specifies the title text of the dialog.

`flag` is used only in Pi/Windows. It may either have the value `FB_SAVE` or `FB_OPEN`, depending on whether the file selection box is being used to save or open a file. It controls the title next if none is supplied, and the label on one of the fields in the dialog. This argument is ignored in Pi/Motif.

The variant `jm_filebox` is a built–in control function. Its first argument is a field name or the name of a JPL variable. The selected file name is copied to this field or variable instead of to the buffer. The `path`, `file_mask`, `title` and `flag` arguments are the same as for `sm_filebox`. To leave an argument out, use `""` in its place. Built–in control functions may be used in control strings and in JPL `call` statements. A menu bar can open a file selection box by calling `jm_filebox` from a control string.

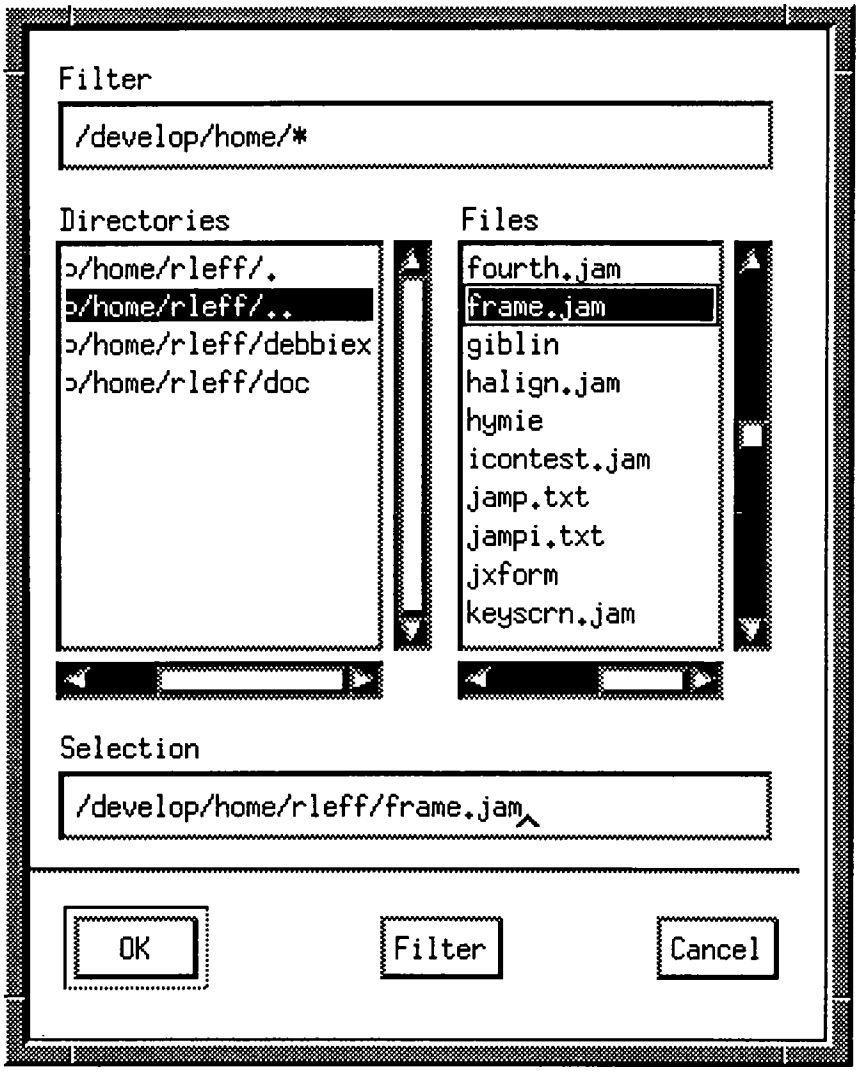


Figure 64: A Motif File Selection Box

The Motif file selection box is illustrated in Figure 64. It is a composite widget consisting of the following:

two text fields: one for a filter and one for the selected file.

The "filter" text field provides a mask for narrowing down the possible file names. The `file_mask` argument supplies the initial filter. It should contain at least one wildcard character. The user may edit the filter.

The "selected file" text field indicates the currently selected file name. The user may also type into this field directly.

two scrolling lists: one list for directories and one for file names.

The user may scroll through the directory list and select a directory by clicking on it once. Clicking twice on a directory updates the file list with the contents of the directory and applies the filter. The `path` argument supplies the initially selected directory.

Clicking once on the file list copies the file name to the selected file field. Clicking twice selects the file.

three push buttons: `OK`, `Filter` and `Cancel`:

`OK` exits the dialog box, copies the full pathname of the selected file to `buffer`, and returns a one. It is up to the developer to provide a properly sized buffer. The buffer's size is indicated by the `length` argument.

`Filter` initiates a directory search, applying the filter to the file list. This has the same result as double clicking on a directory name.

`Cancel` exits the dialog box and returns zero. No text is copied to `buffer`, and the function returns zero.

The Windows file selection dialog is illustrated in Figure 65. It consists of the following:

one text field: this field initially contains the `file_mask`. The user may type another mask into this field, or type in the file name of the selected file. As the user scrolls through the file name list box (see below), the name of the field under the cursor appears in this field.

two list boxes: one for file names and one for directories.

The user may scroll through the directory list and select a directory by clicking on it once. Clicking twice on a directory updates the file list with the contents of the directory and applies the filter. The `path` argument supplies the initially selected directory.

Clicking once on the file list copies the filename to the selected file field. Clicking twice selects the file.

two combo boxes: one for the file type and one for the drive letter. The file type is controlled by a separate function, `sm_filetypes`, available only in P*i*/Windows. The initial drive letter is supplied by the `path` argument.

two push buttons: `OK` and `Cancel`:

`OK` exits the dialog box, copies the full pathname of the selected file to `buffer`, and returns a one. It is up to the developer to provide a properly sized buffer. The buffer's size is indicated by the `length` argument.

`Cancel` exits the dialog box and returns zero. No text is copied to `buffer`, and the function returns zero.

The `flag` argument is used in P*i*/Windows. It may have one of two values:

`FB_OPEN` Use this if the filebox is for opening a file. With this flag, the title defaults to "Open" if no `title` argument is supplied, and the "file types" field has the label "List Files of Type".

`FB_SAVE` Use this if the filebox is for saving a file. With this flag, the title defaults to "Save As" if no `title` is supplied, and the "file types" field has the label "Save File as Type".

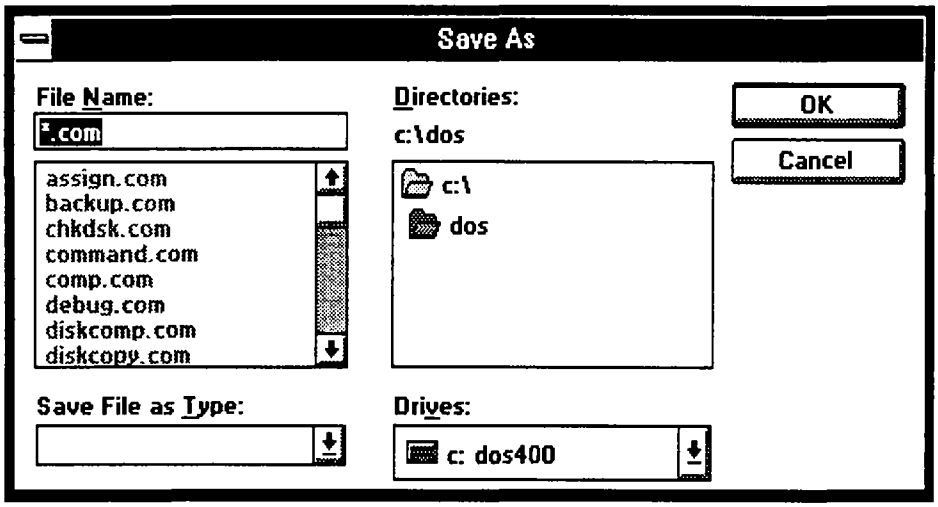


Figure 65: A Windows File Selection Box

RETURNS

1 if the user presses OK. The full pathname of the selected file is copied to the buffer.
0 if the user presses `Cancel`.
–1 if there is a memory allocation error or `buffer` is too small.

RELATED FUNCTIONS

```
sm_filetypes(description, filters);
```

SUPPORTED INTERFACES

Pi/Windows
Pi/Motif

EXAMPLE

```
#include "smdefs.h"
#include "smpi.h"

#define LENGTH 256
char buf [LENGTH];

sm_filebox(buf, LENGTH, "/usr/home/bill", "*.txt", "Bill's Files", 0);
```

sm_filetypes

set up a list of file types for a file selection dialog box

SYNOPSIS

```
#include "smpi.h"

int sm_filetypes(description, filters)
char *description;
char *filters;
```

DESCRIPTION

This function sets up a list of filters for display in the "file type" field of a file selection dialog box under Windows. A file selection dialog is brought up by the routine `sm_filebox`. The file type field contains a list of file types, or masks, that can be set up by the developer. It provides a convenient way for the user to narrow down a directory listing.

`description` is a text string describing a file type. It appears in the list of file types. `filters` is a semicolon separated list of file masks that are included in the particular file type. Each time this function is called, a new `description` and set of `filters` is added to the end of the existing file type list.

To erase the file types list, call `sm_filetypes` with null pointers (or null strings).

This function must be added to the prototyped function list if it is to be called from JPL. In Motif, `sm_filetypes` is ignored.

RETURNS

0 if the description is successfully added to the list.
–1 if there is a memory allocation error.

RELATED FUNCTIONS

```
sm_filebox(buffer, length, path, file_mask, title, flag);
```

SUPPORTED INTERFACES

P*i*/Windows

EXAMPLE

```
#include "smdefs.h"
#include "smpi.h"

/* Clear the file types list, set up two file type filters, and call
 * the filebox routine. */
```

```
#define LENGTH 256
char buf [LENGTH];

sm_filetypes(NULL, NULL);
sm_filetypes("Text files", "*.doc; *.txt");
sm_filetypes("Executables", "*.com; *.exe; *.bat");
sm_filebox(buf, LENGTH, "c:\\", "*.*", "", FB_OPEN);
```

sm_menuinit

initialize menu bar support

SYNOPSIS

```
void sm_menuinit();
```

DESCRIPTION

This routine should be called explicitly only if you are writing a Custom Executive. If you are using the **JAM** Executive, then you simply have to enable support for menu bars in the `main` routine (either `jmain.c` or `jxmain.c`) by setting the appropriate `#define` to `1`. This will cause the `main` routine to call this routine automatically.

If you are writing a Custom Executive and you wish to include menu bar support, you must call this routine. It should be done in the main routine before the call to `initcrt`.

The routine simply sets a global variable to point to a control function. All screen manager functions that need menu bar support check the variable and, if it is non–zero, call indirectly with the request.

If an application is to use keysets in character **JAM** and menu bars in **JAM/P*i***, then the `main` routine should call `sm_skeyinit` before it calls `sm_menuinit`. The second library call will override the first in **JAM/P*i***, but will be ignored in character **JAM**.

If you wish to store menu bars in memory, you must also call `sm_mn_forms`, or set the appropriate `#define` in the `main` routine.

NOTE: Since menu bars and keysets share the same hooks, they may not be used together.

RELATED FUNCTIONS

```
sm_mn_forms();
```

SUPPORTED INTERFACES

P*i*/Windows
P*i*/Motif
P*i*/OPEN LOOK

sm_mn_forms

install menu bars in memory

SYNOPSIS

```
void sm_mn_forms();
```

DESCRIPTION

This routine should be called explicitly only if you are writing a Custom Executive. If you are using the **JAM** Executive, then you simply have to enable support for menu bars in the `main` routine (either `jmain.c` or `jxmain.c`) by setting the appropriate `#define` to `1`. This will cause the `main` routine to call this routine automatically.

If you are writing a Custom Executive and storing menu bars in memory, this routine should be called by the main application program to install the menu bars in memory for use by the screen manager. You must compile menu bars stored in memory into your application and add them to the memory–resident screen list, described in Chapter 9 of the **JAM** *Programmer's Guide*. An alternative to storing menu bars in memory is to open a library of menu bars or to open the menu bars as individual files on disk

A related function, `sm_menuinit`, must also be called in order to initialize menu bar support. To open a menu bar stored in memory, call `sm_d_menu` or `sm_r_menu`.

RELATED FUNCTIONS

```
sm_menuinit();
sm_d_menu(menu, scope);
sm_r_menu(menu_name, scope);
```

SUPPORTED INTERFACES

P*i*/Windows
P*i*/Motif
P*i*/OPEN LOOK

sm_mnadd

add an item to the end of a menu bar

SYNOPSIS

```
#include "smsoftk.h"
#include "smkeys.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mnadd(scope, menu_name, data)
int scope;
char *menu_name;
struct item_data *data;
```

DESCRIPTION

Adds an item at the end of the menu bar specified by `scope` and `menu_name`.

`scope` is one of the mnemonics listed in `smsoftk.h`, and shown in the table below.

Scope	*Description*
KS_FORM	Screen–level menu bar.
KS_APPLIC	Application–level menu bar.
KS_OVERRIDE	Override–level menu bar.
KS_MEMRES	Memory–resident menu bar.
KS_SYSTEM	System–level menu bar.

`menu_name` is the name of the menu as specified in the menu script.

`item_data` is a user–allocated structure that describes the appearance and function of a menu bar item. Its contents are shown in the table below:

Member	*Description*
type	Specifies the type of item. Possible values are: MT_SEPARATOR, MT_TITLE, MT_SUBMENU, MT_KEY, MT_CTRLSTRNG, MT_EDIT, MT_WINDOWS
label	Label text for the item. Text beyond 255 characters is truncated. The label is ignored if type is MT_SEPARATOR. Default is 0.

Member	*Description*
`accel`	Offset of the keyboard shortcut character in the label text string. Default is `-1`.
`key`	Logical key mnemonic. This is used only if `type` is `MT_KEY`. See `smkeys.h` for a listing of valid key mnemonics. Default is `0`.
`submenu`	A text string containing the submenu name. This is used only if `type` is `MT_SUBMENU`. Default is `0`.
`option`	Display options. There are separate display options for separators and text type items. See the table below.

Any structure members that are not relevant to the item should have the default value, namely: `0` for `label`, `key`, and `submenu`; and `-1` for `accel`.

The mnemonics for display options shown in the following table are defined in `smmenu.h`. They are described in detail in the menu bar chapter in section 8.4. Text options may be bitwise or'ed together; separator options may not.

Text Item Options	*Value*	*Separator Options*	*Value*
MO_INDICATOR_ON	0x0200	MO_SINGLE	0x0000
MO_MENUBREAK	0x0400	MO_DOUBLE	0x0001
MO_INDICATOR	0x0800	MO_NOLINE	0x0002
MO_GRAYED	0x1000	MO_SINGLE_DASHED	0x0003
MO_INACTIVE	0x2000	MO_DOUBLE_DASHED	0x0004
MO_SHOWKEY	0x4000	MO_ETCHEDIN	0x0005
MO_HELP	0x8000	MO_ETCHEDOUT	0x0006

RETURNS

0 if there is no error.
–2 if there is no menu bar at this scope.
–3 if menu bars are not supported or `scope` is out of range.
–4 if `menu_name` is not found.
–6 if data in `item_data` is bad.
–7 if there is a malloc error.

RELATED FUNCTIONS

```
sm_mnchange(scope, menu_name, item_no, data);
sm_mndelete(scope, menu_name, item_no);
sm_mnget(scope, menu_name, item_no, data);
sm_mninsert(scope, menu_name, item_no, data);
sm_mnitems(scope, menu_name);
sm_mnnew(scope, menu_name);
```

SUPPORTED INTERFACES

Pi/Windows
Pi/Motif
Pi/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"
#include "smkeys.h"

...

struct item_data *data;
data = ( struct item_data * ) malloc( sizeof( struct item_data ) );

/* Call sm_d_menu w/ a disk resident menu and KS_FORM.
 * Call sm_mnadd to add a title for submenu.
 */

sm_r_menu("mymenu.bin", KS_FORM);
data->type = MT_TITLE;
data->label = "Submenu";
data->accel = -1;
data->key = 0;
data->submenu = 0;
data->option = MO_INDICATOR_ON;
sm_mnadd(KS_FORM, "Submenu0", data);

...
```

sm_mnchange

alter a menu bar item

SYNOPSIS

```
#include "smsoftk.h"
#include "smkeys.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mnchange(scope, menu_name, item_no, data)
int scope;
char *menu_name;
int item_no;
struct item_data *data;
```

DESCRIPTION

Change the data associated with the menu bar item specified by `item_no`, `menu_name` and `scope`, to the data contained in the `item_data` structure. `item_data` is a user-allocated structure that describes the appearance and function of a menu bar item. See `sm_mnadd` for details on the `item_data` structure and a listing of the various scopes. The first item on a menu is `item_no` zero.

Use this routine, for example, to grey out or check an item.

RETURNS

0 if there is no error.
–2 if there is no menu bar at this `scope`.
–3 if menu bars are not supported or `scope` is out of range.
–4 if `menu_name` is not found.
–5 if `item_no` is not found.
–6 if data in `item_data` is bad.
–7 if there is a malloc error.

RELATED FUNCTIONS

```
sm_mnadd(scope, menu_name, data);
sm_mndelete(scope, menu_name, item_no);
sm_mnget(scope, menu_name, item_no, data);
sm_mninsert(scope, menu_name, item_no, data);
sm_mnitems(scope, menu_name);
sm_mnnew(scope, menu_name);
```

SUPPORTED INTERFACES

P*i*/Windows
P*i*/Motif
P*i*/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"
#include "smkeys.h"

...

/* menu file stored in memory */
extern char mymenu[];

...

struct item_data *data;
data = ( struct item_data * ) malloc( sizeof( struct item_data ) );

/* Call sm_r_menu w/ a disk resident menu and KS_APPLIC.
 * Call sm_mnchange to grey out a menu item in the submenu.
 */

sm_r_menu("mymenu.bin", KS_APPLIC);
data->type = MT_KEY;
data->label = "NewItem";
data->accel = 3;
data->key = PF1;
data->submenu = 0;
data->option = MO_GRAYED|MO_SHOWKEY;
sm_mnchange(KS_APPLIC, "Submenu0", 0, data);

...
```

sm_mndelete

delete a menu bar item

SYNOPSIS

```
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mndelete(scope, menu_name, item_no)
int scope;
char *menu_name;
int item_no;
```

DESCRIPTION

Delete the item specified by `item_no`, `menu_name`, and `scope` from the menu bar. The first item on a menu is `item_no` zero.

RETURNS

0 if there is no error.
–2 if there is no menu bar at this `scope`.
–3 if menu bars are not supported or `scope` is out of range.
–4 if `menu_name` is not found.
–5 if `item_no` is not found.

RELATED FUNCTIONS

```
sm_mnadd(scope, menu_name, data);
sm_mnchange(scope, menu_name, item_no, data);
sm_mnget(scope, menu_name, item_no, data);
sm_mninsert(scope, menu_name, item_no, data);
sm_mnitems(scope, menu_name);
sm_mnnew(scope, menu_name);
```

SUPPORTED INTERFACES

Pi/Windows
Pi/Motif
Pi/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"
```

```
...

int count;

/* Delete the last item from the application menu called "customer" */

if ((count = sm_mnitems( KS_APPLIC, "customer" )) > 0)
  sm_mndelete( KS_APPLIC, "customer", count );

...
```

sm_mnget

get menu bar item information

SYNOPSIS

```
#include "smsoftk.h"
#include "smkeys.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mnget(scope, menu_name, item_no, data)
int scope;
char *menu_name;
int item_no;
struct item_data *data;
```

DESCRIPTION

Get the specified menu bar item's data. Given the `menu_name` (as given in the menu script) and an `item_no`, this function fills the fields in the `item_data` structure with the associated data for that item. The first item on a menu is `item_no` zero. Note that you must create buffers for the label and submenu elements of the structure that are large enough to hold the label and submenu names (see the example below). The maximum length is 255 characters. See `sm_mnadd` for details on the `item_data` structure and a listing of the various scopes.

RETURNS

0 if there is no error.
–2 if there is no menu bar at this `scope`.
–3 if menu bars are not supported or `scope` is out of range.
–4 if `menu_name` is not found.
–5 if `item_no` is not found.

RELATED FUNCTIONS

```
sm_mnadd(scope, menu_name, data);
sm_mnchange(scope, menu_name, item_no, data);
sm_mndelete(scope, menu_name, item_no);
sm_mninsert(scope, menu_name, item_no, data);
sm_mnitems(scope, menu_name);
sm_mnnew(scope, menu_name);
```

SUPPORTED INTERFACES

Pi/Windows
Pi/Motif
Pi/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smmach.h"
#include "smmenu.h"
#include "smsoftk.h"

...

/* menu file stored in memory */
extern char mymenu[];

...

char buf1[100], buf2[100];

struct item_data *data;

data = ( struct item_data * ) malloc( sizeof( struct item_data ) );

data->label = buf1;
data->submenu = buf2;

/*  Call sm_r_menu with a disk resident menu.
 *  Call sm_mnget to get an override-level menu bar item.
 */

sm_r_menu("mymenu.bin", KS_OVERRIDE);
sm_mnget(KS_OVERRIDE, "Main", 0, data );

...
```

sm_mninsert

insert a new menu bar item

SYNOPSIS

```
#include "smsoftk.h"
#include "smkeys.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mninsert(scope, menu_name, item_no, data)
int scope;
char *menu_name;
int item_no;
struct item_data *data;
```

DESCRIPTION

Insert a new menu bar item before the menu item specified by `item_no`, `menu_name`, and `scope`, using the data in the menu bar structure `item_data`. `item_data` is a user–allocated structure that describes the appearance and function of a menu bar item. See `sm_mnadd` for details of the `item_data` structure and a listing of the various scopes. The first item on a menu is `item_no` zero.

RETURNS

0 if there is no error.
–2 if there is no menu bar at this `scope`.
–3 if menu bars are not supported or `scope` is out of range.
–4 if `menu_name` is not found.
–5 if `item_no` is not found.
–6 if data in `item_data` is bad.
–7 if there is a malloc error.

RELATED FUNCTIONS

```
sm_mnadd(scope, menu_name, data);
sm_mnchange(scope, menu_name, item_no, data);
sm_mndelete(scope, menu_name, item_no);
sm_mnget(scope, menu_name, item_no, data);
sm_mnitems(scope, menu_name);
sm_mnnew(scope, menu_name);
```

SUPPORTED INTERFACES

P*i*/Windows
P*i*/Motif
P*i*/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"
#include "smkeys.h"

...

struct item_data *data;

data = ( struct item_data * ) malloc( sizeof( struct item_data ) );

/* Call sm_r_menu w/ a disk resident menu and KS_FORM.
 * Call sm_mninsert to insert a submenu.
 */

sm_r_menu("mymenu.bin", KS_FORM);
data->type = MT_SUBMENU;
data->label = "NewItem";
data->accel = 3;
data->key = 0;
data->submenu = "Submenu1";
data->option = MO_INDICATOR;
sm_mninsert(KS_FORM, "Main", 1, data);

...
```

sm_mnitems

get the number of items on a menu bar

SYNOPSIS

```
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mnitems(scope, menu_name)
int scope;
char *menu_name;
```

DESCRIPTION

Returns the number of items on the menu bar specified by `menu_name` and `scope`. Refer to `sm_mnadd` for a list of values for `scope`. When referring to items in related functions, the first item on a menu is item number zero.

RETURNS

–2 if there is no menu at this `scope`.
–3 if menu bars are not supported or `scope` is out of range.
–4 if `menu_name` is not found.
otherwise the number of items on the menu bar is returned.

RELATED FUNCTIONS

```
sm_mnadd(scope, menu_name, data);
sm_mnchange(scope, menu_name, item_no, data);
sm_mndelete(scope, menu_name, item_no);
sm_mnget(scope, menu_name, item_no, data);
sm_mninsert(scope, menu_name, item_no, data);
sm_mnnew(scope, menu_name);
```

SUPPORTED INTERFACES

Pi/Windows
Pi/Motif
Pi/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smmach.h"

...
```

```
int ret;

/* Call sm_r_menu w/ a disk resident menu and KS_OVERRIDE.
 * Call sm_mnitems to get the number of items on the menu bar, and
 * place the number in the current field.
 */

sm_r_menu("mymenu.bin", KS_OVERRIDE);
ret = sm_mnitems(KS_OVERRIDE, "Main");
   sm_n_itofield( "number", ret );

...
```

sm_mnnew

create a new menu bar by name

SYNOPSIS

```
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

int sm_mnnew(scope, menu_name)
int scope;
char *menu_name;
```

DESCRIPTION

This routine creates a new submenu in the menubar structure at the specified `scope`. Refer to `sm_mnadd` for a list of values for `scope`. This routine does *not* add an item for the submenu to the top–level menu bar, it simply makes the new submenu available for adding items to, via `sm_mnadd` or `sm_mninsert`. After the new submenu is fleshed out, an entry for it can added to an existing menu or submenu, also via `sm_mnadd` or `sm_mninsert`.

RETURNS

0 if there is no error.

–2 if there is no menu bar at the specified `scope`.

–3 if menu bars are not supported or `scope` is out of range.

–4 if `menu_name` is invalid or already exists.

–7 if there is a malloc error.

RELATED FUNCTIONS

```
sm_mnadd(scope, menu_name, data);
sm_mnchange(scope, menu_name, item_no, data);
sm_mndelete(scope, menu_name, item_no);
sm_mnget(scope, menu_name, item_no, data);
sm_mninsert(scope, menu_name, item_no, data);
sm_mnitems(scope, menu_name);
```

SUPPORTED INTERFACES

Pi/Windows
Pi/Motif
Pi/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"
#include "smkeys.h"

...

int ret;
struct item_data *data;

data = ( struct item_data * ) malloc( sizeof( struct item_data ) );

/* Call sm_r_menu w/ a disk resident menu and KS_OVERRIDE.
 * Call sm_mnnew to create a new menu bar .
 * Call sm_mnadd to add items to it and finally add this new menu
 * to the menu displayed as a submenu.
 */

sm_r_menu("main.bin", KS_OVERRIDE);
ret = sm_mnnew(KS_OVERRIDE, "NewItem");
   if ( ret == 0 )
   {
     data->type = MT_TITLE;
     data->label = "Submenu";
     data->accel = -1;
     data->key = 0;
     data->submenu = 0;
     data->option = MO_INDICATOR_ON;

     sm_mnadd(KS_OVERRIDE, "NewItem", data);

     data->type = MT_SUBMENU;
     data->label = "I";
     data->accel = 0;
     data->key = 0;
     data->submenu = "Submenu1";
     data->option = MO_INDICATOR;

     sm_mnadd(KS_OVERRIDE, "NewItem", data);

     data->type = MT_SUBMENU;
     data->label = "NewItem";
     data->accel = 3;
     data->key = 0;
     data->submenu = "NewItem";
     data->option = MO_INDICATOR;

     sm_mnadd(KS_OVERRIDE, "Main", data);
   }
...
```

sm_r_menu

read and display a menu bar from memory, a library or disk

SYNOPSIS

```
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

int sm_r_menu(menu_name, scope)
char *menu_name;
int scope;
```

DESCRIPTION

The parameter menu_name is the name of the menu bar. This name is sought first in the memory–resident screen list, next in any open libraries and finally on disk in the directories specified by the argument to sm_initcrt and by SMPATH. Screens and menu bars may be mixed in the screen list and in libraries.

scope is one of the mnemonics listed in smsoftk.h and shown in the table below.

Scope	*Description*
KS_FORM	Screen–level menu bar.
KS_APPLIC	Application–level menu bar.
KS_OVERRIDE	Override–level menu bar.
KS_MEMRES	Memory–resident menu bar.
KS_SYSTEM	System–level menu bar.

If there is currently a menu bar with the specified scope the name of that menu bar is compared with menu_name. If they are the same, the routine returns immediately. Thus to refresh a menu bar with a new copy from disk, call sm_c_menu first.

If scope is KS_OVERRIDE, the currently displayed menu bar is saved in a stack (o_stack). When the override menu bar closes, the saved menu bar is restored. This stacking is performed only for a scope of KS_OVERRIDE. This scope is used for help screens, zoom windows, etc. The stack is fixed at 10 deep.

If scope is KS_MEMRES, the menu bar is read and added to the stack of memory–resident menu bars for use as external menus.

For all other scopes, the menu bar is read and installed. The old menu bar at this scope, if any, is freed. If the menu bar at this scope is currently displayed, it must be refreshed. This fact is marked and the actual refresh is performed at the next delayed write.

RETURNS

0 if no error occurred during display of the menu bar.
–1 if the format is incorrect (not a menu bar).
–2 if menu_name is not found.
–3 if menu bars are not supported or the scope is out of range.
–4 if there is a read error.
–5 if there is a malloc failure.

In the case of an error the previously displayed menu bar remains displayed.

For all errors except –3 a message is posted to the operator.

RELATED FUNCTIONS

```
sm_c_menu(scope);
sm_d_menu(menu, scope);
```

SUPPORTED INTERFACES

P*i*/Windows
P*i*/Motif
P*i*/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smsoftk.h"
#include "smmach.h"
#include "smmenu.h"

...

/* Read in the company menu and display it at the form level. */

sm_r_menu( "company.bin", KS_FORM );

...
```

sm_translatecoords

translate screen coordinates to display coordinates

SYNOPSIS

```
#include "smpi.h"

int sm_translatecoords(column, line, column_ptr, line_ptr)
int column;
int line;
int *column_ptr;
int *line_ptr;
```

DESCRIPTION

Translates the **JAM** `line` and `column` relative to a screen, into pixel line and column relative to the upper left hand corner of the drawing area. `line` and `column` are zero based. This function in conjunction with `sm_drawingarea` is useful when placing objects such as bitmapped graphics or custom widgets on a **JAM** screen. Refer to the source listing for the pie chart demonstration provided with **JAM/P*i*** for a detailed example of how to import graphics and use these functions.

RETURNS

The pixel coordinates are placed in the integers referenced by `*column_ptr` and `*line_ptr`.

The function also returns:
–1 if the `line` or `column` is out of range;
0 otherwise.

RELATED FUNCTIONS

```
sm_drawingarea();
```

SUPPORTED INTERFACES

P*i*/Windows
P*i*/Motif
P*i*/OPEN LOOK

EXAMPLE

```
/* The following program illustrates how to use sm_drawingarea and
 * sm_translatecoords to display a bitmap on the current JAM screen in
 * Pi/Windows.
 */
```

```
#include <windows.h>
#include <smdefs.h>


void DrawBitmap(HDC hdc, HBITMAP hBitmap, short xStart, short yStart);

int
JAM_display_bitmap( char *bitmap_name, int line, int col )
{
    HWND hwnd;
    HDC hdc;
    HBITMAP hBitmap;
    int pixel_line;
    int pixel_col;

    hwnd = sm_drawingarea();
    hdc = GetDC( hwnd );

    hBitmap = LoadBitmap( GetWindowWord( hwnd, GWW_HINSTANCE ),
                bitmap_name );
    if (hBitmap == NULL)
        {
        char buf[100];

        sprintf( buf, "JAM_display_bitmap: no such bitmap '%s'",
            bitmap_name );
        sm_emsg( buf );
        return( -1 );
        }

    if (sm_translatecoords( col, line, &pixel_col, &pixel_line ) < 0)
        {
        char buf[100];

        sprintf( buf, "JAM_display_bitmap: invalid line/column: %d/%d",
             line, col );
        sm_emsg( buf );
        return( -1 );
        }

    DrawBitmap( hdc,
            hBitmap,
            (short) pixel_col,
            (short) pixel_line );

    DeleteObject( hBitmap );
    ReleaseDC( hwnd, hdc );
    return( 0 );
    }

void
DrawBitmap( HDC hdc, HBITMAP hBitmap, short xStart, short yStart )
```

```
{
    BITMAP bm;
    HDC hdcMem;
    DWORD dwSize;
    POINT ptSize, ptOrg;

    hdcMem = CreateCompatibleDC( hdc );
    SelectObject( hdcMem, hBitmap );
    SetMapMode( hdcMem, GetMapMode( hdc ) );

    GetObject( hBitmap, sizeof( BITMAP ), (LPSTR) &bm );
    ptSize.x = bm.bmWidth;
    ptSize.y = bm.bmHeight;
    DPtoLP( hdc, &ptSize, 1 );

    ptOrg.x = 0;
    ptOrg.y = 0;
    DPtoLP( hdcMem, &ptOrg, 1 );

    BitBlt( hdc, xStart, yStart, ptSize.x, ptSize.y, hdcMem, ptOrg.x,
       ptOrg.y, SRCCOPY );
    DeleteDC( hdcMem );
    }
```

sm_widget

get the widget id of a widget

SYNOPSIS

W

```
#include "mswin.h"
HWND sm_widget(field_number);
HWND sm_n_widget(field_name);
HWND sm_e_widget(field_name, element);
```

M O

```
Widget sm_widget(field_number);
Widget sm_n_widget(field_name);
Widget sm_e_widget(field_name, element);
```

DESCRIPTION

Provides the widget id of (or handle to) a widget, given a field number, field name, or field name and element number. The widget id is necessary for GUI function calls where you wish to interact directly with a particular widget.

M

A series of tables in Chapter 7 list the widgets used in P*i*/Motif. Widgets with an asterisk next to them in the tables are the widgets returned by `sm_widget`.

Note that for scale widgets, list box widgets and multiline text widgets, the widget id returned by `sm_widget` is that of the scroll bar. Use `XtParent` to obtain the id of the scale, list box or multiline text widget.

O

A series of tables in Chapter 7 list the widgets used in P*i*/OPEN LOOK. Widgets with an asterisk next to them in the tables are the widgets returned by `sm_widget`.

RETURNS

Returns NULL if there is no such widget.
Otherwise:

A handle to the widget.

The widget id as a Widget.

RELATED FUNCTIONS

```
sm_drawingarea();
```

SUPPORTED INTERFACES

Pi/Windows
Pi/Motif
Pi/OPEN LOOK

sm_win_shrink

trim the current screen

SYNOPSIS

```
#include "smpi.h"

int sm_win_shrink(void)
```

DESCRIPTION

This routine trims all space on a screen to the right of the rightmost widget and below the bottommost widget. It does not change the number of **JAM** lines and columns. It is primarily useful when `hoff` or `voff` extensions are heavily used to reposition fields. Call `sm_adjust_area()` to restore a screen to its original size.

SUPPORTED INTERFACES

P*i*/Motif
P*i*/OPEN LOOK

12.2
UTILITIES

Two utilities are provided for creating menu bars. The first, `menu2bin`, converts an ASCII menu script into a binary menu file. The second, `kset2mnu`, converts a **JAM** keyset into an ASCII menu script. For detailed instructions on creating menu bar scripts refer to Chapter 8.

menu2bin

convert ASCII menu scripts to binary format

SYNOPSIS

```
menu2bin [-pv] [-e ext] menufile...
```

OPTIONS

`-p` Places the binary files in the same directories as the input files.

`-v` Lists the name of each input file as it is processed.

`-e` Appends ***ext*** to the output file name. The default extension is `bin`.

DESCRIPTION

The `menu2bin` utility converts ASCII menu scripts into binary format for use by **JAM/P***i* applications in place of keysets. Menu scripts are created as text files. Refer to section 8.4 for instructions on creating a menu script.

To store a menu file in memory, first run the binary file produced by this utility through the `bin2c` utility to produce a program source file; then compile that file and link it with your program and add it to the memory-resident screen list (see Chapter 9 of the **JAM** *Programmer's Guide*). The extended library routines `sm_d_menu` and `sm_r_menu` can display menu bars stored in memory.

Menu binary files can be placed in libraries with the `formlib` utility. Refer to the **JAM** *Utilities Guide* for more information.

ERRORS

`Too many menu definitions. Max is 128.`
Cause: Only 128 menu definitions may be included in one menu script.

`Too many item definitions. Max is 128.`
Cause: Only 128 item specifications may be included in one menu definition.

`Cannot create '%s'`
`Error writing '%s'`
Cause: An output file could not be created, due to lack of permission or perhaps lack of disk space.
Corrective action: Correct the file system problem and retry the operation.

`Neither '%s' nor '%s' found.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence and permissions of the file in question.

```
Error in '%s' line '%d'
```

followed by one of the following:

```
Expected left brace '{' after menu name.
No right brace '}' found before EOF.
No menu name specified.
Expected quoted item label.
Missing action.
Unknown action '%s'.
Unknown option '%s'.
No key specified.
Bad key '%s'.
Bad escape sequence '%s'.
Undefined submenu '%s'.
More than one option of this type (%s).
More than one accelerator character assigned.
Accelerator character at end of string - Ignored.
Menu '%s' is on menu bar so cannot be used as submenu.
```

Cause: The syntax of your script on the specified line is incorrect.
Corrective action: Find the error on the line specified and correct it. Refer to section 8.4 for a description of the proper syntax, and a sample menu script.

kset2mnu

convert keysets into ASCII menu scripts

SYNOPSIS

```
kset2mnu [-pv] [-e ext] keyset...
```

OPTIONS

`-p` Places the binary files in the same directories as the input files.

`-v` Lists the name of each input file as it is processed.

`-e` Appends ***ext*** to the output file name. The default extension is `mnu`.

DESCRIPTION

The `kset2mnu` utility converts keysets into menu scripts. The file is converted according to the following rules:

- The first row in the keyset becomes the top–level menu.
- Subsequent rows become submenus. Submenus are named "Row***x***", where ***x*** is the row number.
- The `SFT`***x*** key (goto row ***x***) becomes an entry for the submenu named Row***x***.
- The `SFTN` (next row) and `SFTP` (previous row) keys become entries for the submenus named Row***{i+1}*** or Row***{i–1}***, where ***i*** is the current row.

The menu script created by the utility is an ASCII text file. Refer to section 8.4 for an explanation of the structure of a menu script. You may wish to edit the script produced by the conversion utility to make your converted menu bars more like standard menu bars. While keysets often have direct actions in their first row, menu bars usually have no direct actions on the top level menu, only entries for submenus.

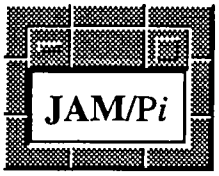
Once you are happy with the contents and display options of your script, run the script through the `menu2bin` utility and install it in your application.

ERRORS

`Soft key '%s' designates a nonexistent submenu.`
Cause: The keyset contains a `SFT`*n* key for a row that does not exist.
Corrective action: Remove the offending key from the keyset and reconvert it.

`Neither '%s' nor '%s' found.`
Cause: An input file was missing or unreadable.
Corrective action: Check the spelling, presence, and permissions of the input file.

`Cannot create '%s'`
`Error writing '%s'`
Cause: An output file could not be created, due to lack of permission or disk space.
Corrective action: Correct the file system problem and retry the operation.



Appendix A
Terminology

The following terms are used throughout the manual. Some of these terms are defined more rigorously in the *Glossary* Appendix to Volume 1 of the **JAM** Manual.

General Terms

character JAM — The **JAM** product for character–based terminals.

initialization file — A text file containing default specifications for the appearance and behavior of Microsoft Windows applications. The `jam.ini` and `win.ini` files are examples of initialization files. Contrast with *resource file* in Motif.

JAM/Pi — The **JAM**/Presentation *interfaces* for Windows and Motif.

Motif — An X widget set created by the Open Software Foundation. Motif also includes an Application Program Interface (API), and a window manager.

OPEN LOOK — An X widget set created by UNIX System Laboratories. OPEN LOOK also includes an Application Program Interface (API), and a window manager.

Pi/Motif — The **JAM**/Presentation *interface* for Motif.

Pi/OPEN LOOK — The **JAM**/Presentation *interface* for OPEN LOOK.

Pi/Windows — The **JAM**/Presentation *interface* for Microsoft Windows.

resource file — A text file containing default specifications for the appearance and behavior of Motif applications. The `.Xdefaults` file, and the `XJam` file are examples of resource files. Individual items in the file are called resources. Contrast with *initialization file* in Windows.

Windows — The Microsoft Windows Graphical User Interface.

Terms Relating to Screens

active screen — The **JAM** screen that is currently accepting input.

base window — An optional window in P*i*/Motif that contains only a status line, keyset and menu bar.

display — The physical screen of the terminal or monitor.

focus — The GUI window that the GUI is sending keyboard input to has focus. This may or may not be the active screen.

form — A **JAM** form.

frame — The area on the display within which **JAM** operates under the Microsoft Windows Multiple Document Interface.

GUI window — A region on the display that may be created by an application. **JAM** screens appear within GUI windows.

screen — General term for a **JAM** form or **JAM** window.

window — A **JAM** window. Windows may be stacked or sibling.

Terms Relating to Items on Screens

bounce bar — A highlighted bar that indicates a selection on a menu.

control — The Windows equivalent of a widget. This document uses the term *widget* in favor of the term *control.*

fixed width font — A font in which each character has the same width, determined by the point size of the font. Most standard terminals use fixed width fonts. `This sentence is set in a fixed width font.`

menu — A **JAM** on–screen menu, consisting of a field or set of fields with the menu edit.

menu bar — The list of pull–down headings that appears on certain screens, directly below the title bar. Some menu bars appear in the base window or frame, while others may be local to a **JAM** screen.

proportional font — A font in which the widths of the characters vary. Proportional fonts are more readable than fixed width fonts, and they look more elegant. The sentence you are reading is set in a proportional font.

scroll bar A widget that is used to scroll the information in a screen or widget. Scroll bars may be horizontal or vertical. A scroll bar usually has an outward pointing arrow at either end and an elevator (also called a thumb, or scroll box) that moves along within the bar, indicating which portion of the screen is visible. Under Motif, the size of the thumb also indicates how much of the screen is visible. The appearance and functionality of scroll bars are determined by the GUI.

widget A GUI object. GUI applications are built from widgets. Some widgets are used as to interact with an application, while others are for display only. Widgets are created in a hierarchical (parent/child) fashion.. **JAM** fields and groups and display text become widgets in **JAM/P*i***. Widgets are called *controls* in MS Windows. This document uses the term *widget* in favor of the term *control.*

INDEX

NOTE: Italicized page references (eg.— Array, *17*) indicate figures.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

JAM

PL/1 Programmer's Guide for Stratus

This is the PL/1 Programmer's manual for **JAM** Release 5. It is as accurate as possible at this time; however, both this manual and **JAM** itself are subject to revision.

Please send suggestions and comments regarding this document to:

Technical Publications Manager
JYACC, Inc.
116 John Street
New York, NY 10038

(212) 267–7722

A Note To Language Interface Users

JYACC makes every effort possible to design language interfaces that duplicate the original C Programmers Library. However, due to differences among various programming languages, an exact one to one correspondence is not always possible. In some cases, routines contained in the C version have been replaced with other routines designed to take advantage of a particular programming language's features.

Please note that your interface contains intentionally undocumented routines. Some of these routines are no longer part of JAM, having been replaced by more efficient routines, and are included only for backward compatibility with applications created with earlier versions of JAM. The rest are internal routines and are not intended to be directly accessed by developers.

A Note To Non–UNIX Users

Throughout the manual, a forward slash (/) has been used to indicate a subdirectory. For example,

```
/usr/local/file
```

means that `file` is a file in the directory `local` which is in turn a sub–directory of `usr`, which is not the root directory.

TABLE OF CONTENTS

Chapter 1.
Introduction

This document is intended for JAM Programmers. We discuss the development and creation of executable JAM programs incorporating the Screen Manager, developer–written hook functions, and the JAM Executive. We will briefly touch on how custom executives may be written. Finally, there is a comprehensive reference of JAM library functions.

Discussions on the creation of JAM screens, data dictionaries, and keysets are found in the Author's Guide. JPL is fully documented in the JPL Programmer's Guide.

This document assumes that the reader has previously read the JAM Development Overview and the Author's Guide. The Development Overview is particularly important as the major architectural components of JAM are explained there in detail.

JAM is written in C, and the C programming interface and libraries are distributed with every license. This PL/1 language interface document is an adaptation of the JAM C Programmer's Guide.

You will need to program in PL/1 (or some other supported third–generation language) to accomplish the following tasks:

- To customize JAM to your environment or application by modifying the main program provided in source form with the product.
- To write hook functions that do application–specific and back–end processing during the execution of the application.
- To take full control of the application by writing an application–specific executive[1].
- To create executable JAM Programs.

As discussed in detail in the Development Overview, JAM Applications consist of screens, a data dictionary, hook functions, and an executable program. The creation of

1. It is strongly recommended that the JAM Executive be used in all but the most unusual of circumstances. A comparison of the JAM Executive with your own executive is presented in the Development Overview.

screens and data dictionaries is discussed in the Author's Guide. JPL programming is discussed in the JPL Programmer's Guide. In this chapter, we discuss how to create a JAM program. Compilation and linking are specific to platforms and operating systems and are discussed in the Installation Guide.

Two different versions of an application can be created with JAM. The Application Executable is the program delivered to the end-user to control the run time application. The JAM Authoring Executable is used to create application components and test the application during development. Only the JAM Authoring Executable will grant user access to the Screen Editor, the Data Dictionary Editor, and the Keyset Editor. The JAM Authoring Executable can only be used for the testing of applications that use the JAM Executive.

1.1. APPLICATION EXECUTABLE

Application Executable programs fall into two categories: those that use the JAM Executive to manage the flow of control from screen to screen, and those that use an application-specific executive. We discuss both of these approaches in the sections that follow.

1.1.1. Applications Using the JAM Executive

In applications that use the JAM Executive, most of the control flow is encapsulated in the screens. The majority of the PL/1 programming task is to write hook functions (section 2. page 7) that are called by the Screen Manager or by the JAM Executive when certain events occur.

Applications that use the JAM Executive will need to be linked with the PL/1 interface library `xif`, the Screen Manager library `sm`, the JAM Executive library `jm`, and, in general, the standard math library on your system.

NOTE: Refer to the Stratus Software Release Bulletin for specifics of the VOS library setup.

JYACC provides the main routine source code for applications that use the JAM Executive in a file called `jmain.pl1`. This routine performs various necessary initializations before calling the function that starts up the JAM Executive. You may want to modify this code to change JAM's default behavior.

112.

Applications Using a Custom Executive

In rare cases, a developer may choose to write a custom executive, one that is specific to a particular application. In custom executives, no library functions specific to the JAM Executive should be used. The JAM Executive functions may only be used in applications using the JAM Executive — they are listed in section 11.15. on page 85. .

Applications that do not use the JAM Executive should be linked with the PL/1 interface library `xif`, the Screen Manager library `sm`, and, in general, the standard math library. If the LDB is needed, the JAM Executive library `jm` should also be linked in, but it is important the application not call any JAM Executive routines.

The "sample" application provided with JAM is a simple example of an application using a custom executive.[2] This application brings up a screen on which the end–user can enter some account data, and then save the data and call it up again. There is a help screen, tied to one of the function keys, which is implemented as a memory–resident screen, and a hook written function that verifies the area code. The discussion below outlines the basic steps that a custom executive should perform, using `sample.pl1` as an example.

To follow this discussion, you should either print this file out, or call it up in an editor. Refer to the Stratus Software Release Bulletin for the location of `sample.pl1`. The hook function AREACODE can be found in the same file.

Header Files

JAM user defines are included as necessary, depending on the library routines utilized in the program. The documentation for each library routine indicates which, if any, header files are required.

Declarations

A memory–resident screen is declared at the top of the program along with whatever variables are necessary.

Screen Manager Initialization

After all the header files and declarations at the top of the source module, the Screen Manager and the terminal are first initialized with a call to `xsm_initcrt`. Since an empty string is passed as the argument, the search path for screens is expected to be found in the environment.

2 Note that JPL is available to applications that do not use the JAM Executive Note also that hook functions may be installed and used in applications that do not use the JAM Executive. These applications, however, will not be able to use control strings.

Install Hook Functions

Most Screen Manager hook functions are installed via the `-retain_all` argument to the `bind` command. This is the case for the hook function `areacode`, which is called as a field validation function. For certain types of hook functions, explicit installation is necessary and should occur here—after initialization, but before displaying the first screen. The various types of hook functions and their installation are described in detail in Chapter 2.

Display the Main Form

After initialization is complete, the screen `sample1.frm` is opened as a form with a call to `xsm_r_form`. If an error occurs, the program will terminate.

Activate Screen

`sample1.frm` is activated within a loop. The loop terminates if the user strikes the EXIT key, which causes the routine `xsm_input` to return with the return code EXIT defined in `smkeys.incl.pl1`. The actual data entry, cursor movement, help processing, character edit masking, and validation are handled within `xsm_input`, so the programmer need not be concerned with them. Whenever the user strikes TRANSMIT, EXIT, or some other function key, `xsm_input` returns control to the calling program. In this case, the PF2, PF3 and EXIT keys cause specific actions. All other function keys cause a beep and the while loop to continue, calling `xsm_input` again.

Open a Window

The PF3 key brings up the memory–resident screen that was declared earlier, and then waits for the user to press a key.

Close a Window

During the run of any application, there is always a form displayed. When a new form is displayed, all existing screens are implicitly closed. Windows, however, need to be explicitly closed if the application is to retreat to an underlying screen. After the PF3 window is displayed, when the user strikes a key the program calls `xsm_close_window` to close this window.

Handle Errors

The executive should have a facility to handle errors. The PF2 key triggers a procedure, PROCESS, which opens a window allowing the user to save or read data. While the specifics of this data manipulation are beyond the scope of this introductory discussion, use of the error handling routine `xsm_err_reset`, which displays an error message on the

status line, is illustrated about halfway through the procedure listing. `xsm_err_reset` takes a single string argument, and places that string on the status line. The user is forced to acknowledge the error by striking the space bar[3].

Reset the Terminal

Before the application terminates, it calls `xsm_resetcrt` to reset terminal characteristics to a state expected by the operating system. Here this occurs when the user presses the EXIT key.

1.2.
AUTHORING EXECUTABLE

The Authoring Executable must use the JAM Executive, and may have developer–written hook functions linked in. The main routine for the Authoring Executable is provided in source form in a file called `jxmain.pl1`. You may want to modify that file to change the default behavior of the authoring tool `jxform`. It is strongly suggested that JAM developers read and understand this code, as it is instructive and may help with an understanding of the product.

The compiled Authoring Executable may be called with the optional command–line switch `-e`. This will cause the authoring tool to start up directly within the Screen Editor (as opposed to starting up in application mode).

Authoring executables must be linked with the PL/1 interface library `xif`, the JAM Authoring Library `jx`, the JAM Executive library `jm`, the Screen Manager library `sm`, and, in general, the standard math library. Since these executables are linked with the JAM Authoring Library `jx`, they may not be re–sold or distributed on machines for which there is no software license from JYACC. This restriction applies *only* to Authoring Executables, which are intended for application *development* only.

NOTE: Refer to the Stratus Software Release Bulletin for specifics of the VOS library setup.

3 The developer may change the way messages are acknowledged with the library routine `XSM_OPTION`.

Chapter 2.
Hook Functions

The primary coding task facing JAM programmers is writing hook functions. These functions, which are called by the JAM Executive and by the Screen Manager when certain well–defined events occur, are written in PL/1[5].

In this chapter, we discuss how hook functions are written and installed. They must also be compiled and linked into the JAM Application (or Authoring) Executable: see the Installation Guide for details of that. We also discuss what JAM events have hooks accessible to developers and what arguments are passed to hook functions from any given hook. Finally, we discuss in detail the various types of hook functions, showing examples of some of them, and explaining how they are installed and used.

2.1.
PREPARATION AND INSTALLATION

Hook functions, once properly installed, are called at certain well– defined JAM events. These events are outlined below in section 2.1.1. and discussed in detail later in the chapter.

There are many events that have hooks accessible to developers. JAM passes different arguments to the various hook functions, and interprets the return codes differently for each one. It is important that hook functions process the arguments that are passed correctly, and that they return meaningful codes based on the events to which they are attached.

Hook functions are installed individually, and are called at runtime by JAM when a certain event type occurs. Most hook functions are called by the Screen Manager. However,

5. Hook functions may also be written in C and other third–generation programming languages for which JYACC supports a language interface In particular, Fortran, Cobol and PL/1 are available for JAM on some platforms

the hook functions invoked with control strings are called by the JAM Executive, and will only be accessible to applications using a custom executive through JPL.

2.1.1.

Types of Hook Functions

There are twenty–two installable hook function types, six of which are installed when the application is bound and sixteen of which are installed as individual functions. They are briefly outlined below, and discussed in detail later in the document:

- `FIELD_FUNC`
  These functions are installed using the `-retain_all` argument of the `bind` command. The functions on this list may be designated in the Screen Editor to be called by the Screen Manager as field entry, exit or validation functions for specific fields. The JPL `atch` verb may also be used to access these functions.

- `GROUP_FUNC`
  These functions are installed using the `-retain_all` argument of the `bind` command. These functions may be designated in the Screen Editor to be called by the Screen Manager as group entry, exit or validation functions for specific groups (Radio Buttons and Checklists).

- `SCREEN_FUNC`
  These functions are installed using the `-retain_all` argument of the `bind` command. These functions may be designated in the Screen Editor to be called by the Screen Manager as screen entry or exit functions on particular screens.

- `CONTROL_FUNC`
  These functions are installed using the `-retain_all` argument of the `bind` command. These functions may be entered and invoked from control strings. They are often associated with function keys and menus in the Screen Editor or with the `xsm_putjctrl` library call. The JPL `call` verb can invoke control functions.

- `DFLT_FIELD_FUNC`
  This is an individual function. It is installed using the library routine `xsm_n_uinstall`. Once installed, it is called on entry, exit and validation for all fields.

- `DFLT_GROUP_FUNC`
  Similar to the `DFLT_FIELD_FUNC`, this individual function is called on entry, exit, and validation for all groups.

- DFLT_SCREEN_FUNC
  Individual function called on entry and exit for all screens.

- KEYCHG_FUNC
  Individual function called whenever JAM reads a key from the keyboard. This allows for the application to intercept and process (and possibly translate) keystrokes at the logical key level.

- INSCRSR_FUNC
  Individual function called by JAM whenever the keyboard entry mode toggles between insert and overstrike mode. This allows an application to update the display, if desired, to provide an indication of the new mode. Often used if there is no ability to change cursor styles between insert and overstrike modes.

- CKDIGIT_FUNC
  Individual function called by JAM for check digit validation of numeric fields. Only necessary if the default check-digit algorithm provided with JAM is not sufficient.

- UINIT_FUNC
  Individual function called just before the Screen Manager and the physical display are initialized at the start of the application.

- URESET_FUNC
  Individual function called just after the Screen Manager and the physical display are closed and reset at the end of the application, even if the application aborts ungracefully.

- RECORD_FUNC
  Individual function used to record keystrokes so they can be played back for tutorials or for regression testing.

- PLAY_FUNC
  Individual function used to playback recorded keys.

- AVAIL_FUNC
  Individual function used in advanced record/playback algorithms.

- STAT_FUNC
  Individual function used to intercept JAM status line processing and alter or replace it.

- VPROC_FUNC
  Individual function used to intercept JAM video processing and to alter or replace it.

- BLKDRVR_FUNC
  This is an individual function that acts as a block mode terminal driver. This is discussed in section 10.1.3.

- ASYNC_FUNC
  Individual function called asynchronously when JAM is waiting for keyboard input. This is installed via the library routine `xsm_async`. Often used to poll external systems for mail delivery or the availability of data over a communications line.

2.1.2.
Installing Functions

As mentioned above, certain hook functions must be installed explicitly with the library routines `xsm_n_uinstall` or `xsm_async`, others are installed using the `-retain_all` argument of the `bind` command.

`xsm_n_uinstall` is called with three arguments. The first argument identifies the type of function being installed, and may be one of the following values:

```
UINIT_FUNC          CKDIGIT_FUNC        STAT_FUNC
URESET_FUNC         BLKDRVR_FUNC        DFLT_FIELD_FUNC
VPROC_FUNC          PLAY_FUNC           DFLT_SCREEN_FUNC
KEYCHG_FUNC         RECORD_FUNC         DFLT_GROUP_FUNC
INSCRSR_FUNC        AVAIL_FUNC
```

The second argument is the name of the function. The third argument identifies the language. This argument should be 1 for all programming languages except C.

`xsm_async` is used exclusively for installing asynchronous functions. It takes as arguments the address of the function and a timeout period.

The other function types, which are installed via the `-retain_all` argument to the `bind` command, are the following:

```
FIELD_FUNC
SCREEN_FUNC
CONTROL_FUNC
GROUP_FUNC
```

2.2.
WRITING HOOK FUNCTIONS

Arguments passed to hook functions and return values received from hook functions vary from hook to hook. In this section, we discuss the various JAM hooks in detail.

2.2.1.

Field Functions

The Screen Manager will call field functions, if specified, on field entry, field exit, and field validation. Calls to field entry and field exit functions are guaranteed to be paired for any given field.

A single default field function may also be installed. It will be invoked on entry, exit, and validation for every field. The default field function must be installed explicitly as via `xsm_n_uinstall`.

JPL procedures may be directly specified as field functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl fieldfunc`.

Field Function Invocation

Field functions are called for field entry whenever the cursor enters a field, including when the field containing the cursor is activated by virtue of an overlying window being closed. Field functions are called for field exit whenever the cursor leaves a field, including when the field is exited because a window is popped up over the existing screen. Field functions are called for validation whenever the field is validated. This occurs at the following times:

- As part of field validation, when you exit the field or scroll to the next occurrence by filling it or by hitting TAB or RETURN key. The BACK-TAB and arrow keys do not normally cause validation. Field functions are called for validation only after the field's contents pass all other validations for the field.
- As part of screen validation when the XMIT key is struck.
- When the application code calls library routines for field validation.

Field functions may also be invoked from JPL with the `atch` verb.

For fields that are members of menus, radio buttons, or checklists, the validation function is not called as part of validation. The validation function for such fields is called instead when that field is selected. For checklist fields, the field validation function is also called when the field is deselected.

Field functions specified for field entry via the Screen Editor are invoked after any installed default field function. Field functions specified for field exit or validation via the Screen Editor are called before any installed default field function.

Field Function Arguments

All field functions receive four arguments:

1. The field number as an integer.
2. A buffer containing a copy of the field's contents.
3. The occurrence number of the data as an integer.
4. An integer bitmask containing contextual information about the validation state of the field and the circumstances under which the function was called.

The contextual information in the last parameter includes the following bit masks[7]:

- `VALIDED`
  If this is set (i.e. if the 'bitwise and' of `param4` and `VALIDED` is not zero), the field has passed all its validations and has not been modified since.
- `MDT`
  If this is set (i.e. if the 'bitwise and' of `param4` and `MDT` is not zero), the field data has been changed either from the keyboard or from the application code since the current screen was opened[8]. JAM never clears this bit. The application code may clear it directly with the `xsm_bitop` library routine.
- `K_ENTRY`
  If set (i.e. if the 'bitwise and' of `param4` and `K_ENTRY` is not zero), the field function was called on field entry.
- `K_EXIT`
  If set (i.e. if the 'bitwise and' of `param4` and `K_EXIT` is not zero), the field function was called on field exit[9].
- `K_EXPOSE`
  If set (i.e. if the 'bitwise and' of `param4` and `K_EXPOSE` is not zero), the field function was called because a window overlying the screen on which the field resides was opened or closed[10].
- `K_KEYS`
  Mask for the bits indicating which keystroke or event caused the field to be entered, exited, or validated. The intersection of this mask and the fourth pa-

NO TAG.

The example field function below contains a procedure called `bitmask` that is useful for checking whether a particular flag (bit location in a binary value) is set. Source code for this procedure can also be found in the `sample` application provided with JAM.

8. Note that when the screen is being opened, when the screen entry function modifies data in a field the `MDT` bit is not set. However, when the screen is exposed by virtue of an overlaid window being closed, modification of field data in the screen entry function will cause the `MDT` bit to be set.

9. Note that if neither `K_ENTRY` nor `K_EXIT` are set, the field is being validated.

10 This means that if both `K_ENTRY` and `K_EXPOSE` are set, the field is being exposed. If `K_EXIT` and `K_EXPOSE` are set, the field is being hidden.

rameter to the field function should be tested for equality against one of the six remaining values below:

- K_NORMAL
  If set (i.e. if the 'bitwise and' of param4 and K_KEYS equals 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".

- K_BACKTAB
  If set (i.e. if the 'bitwise and' of param4 and K_KEYS equals K_BACKTAB), the BACKTAB key caused the cursor to enter or exit the field in question.

- K_ARROW
  If set (i.e. if the 'bitwise and' of param4 and K_KEYS equals K_ARROW), an arrow key caused the cursor to enter or exit the field in question.

- K_SVAL
  If set (i.e. if the 'bitwise and' of param4 and K_KEYS equals K_SVAL), the field is being validated as part of screen validation.

- K_USER
  If set (i.e. if the 'bitwise and' of param4 and K_KEYS equals K_USER), the field is being validated directly from the application with the xsm_fval library routine.

- K_OTHER
  If set (i.e. if the 'bitwise and' of param4 and K_KEYS equals K_OTHER), some key other than backtab, arrow or those mentioned as "normal" caused the cursor to enter or exit the field in question.

Field functions are called for validation regardless of whether the field was previously validated. They may test the VALIDED and MDT bits to avoid redundant processing.

Field Function Return Codes

Field functions called on entry or exit should return 0. Field functions called for validation should return 0 if the field contents pass the validation criteria. Any non-zero return code should indicate that the field does not pass validation. If the returned value is 1, the cursor will not be repositioned to the offending field. Any other non-zero return value will cause the cursor to be repositioned to the field that failed the validation. This is useful because when the entire screen is undergoing validation, the field that fails validation

may not be the field where the cursor is.[11]

Example Field Function

The following code illustrates how to interpret the fourth argument passed to a field function, and how to handle errors.

```
%include 'smdefs.incl.pl1';     /* basic JAM user defines */

apfunc1:
    procedure(field_number, field_data, occurrence, misc_bits)
    returns(fixed_binary(31));

    declare field_number              fixed_binary(31);
    declare field_data                char(*) varying;
    declare occurrence                fixed_binary(31);
    declare misc_bits                 fixed_binary(31);
    declare error                     fixed_binary(31);

    if bitmask(misc_bits, VALIDED)
        then return 0

    /* and later...    */
    /* check for error */

    if(error ^= 0)
        then do;
        xsm_gofield(1);
        xsm_quiet_err('Re-enter all data.');
        return(1);
        end;
    return(0);
end apfunc1;

    /* The following procedure checks if a particular flag is set.   */
    /* NOTE:  "unspec" only works on variables. Constants are passed */
    /* into bitmask as parameters, so bitmask will work with them.   */

bitmask:
    procedure(xbits,ybits)
    returns(bit(1));
    declare(xbits,ybits) fixed_binary(31);
    return((unspec(xbits) & unspec(ybits)) ^= '0'b);
end bitmask;
```

11 In many cases, it is better for the field validation function itself to reposition the cursor before displaying an error message, otherwise the error message might be misleading

2.2.2.

Screen Functions

The Screen Manager will call screen functions, if specified, on entry and exit of screens. Calls to screen entry and screen exit functions are guaranteed to be paired for each screen.

A single default screen function may be installed. It will be invoked on entry and exit for every screen. The default screen function is installed as via `xsm_n_uinstall`. Screen functions specified as entry or exit functions for a screen via the Screen Editor are installed via the `-retain_all` argument to the `bind` command. JPL procedures may also be directly specified as screen functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl screenfunc`.

Because of the way LDB processing and form stack handling is done, it is neither recommended nor supported to call any form or window display library routines from screen entry or exit functions. If it is necessary to display windows at screen entry, the library routine `xsm_ungetkey` can be invoked, passing as the argument a function key with a control string that brings up a window.

Screen Function Invocation

Screen functions are called for screen entry whenever a screen is opened. Screen functions are called for screen exit whenever a screen is closed. Optionally, screen functions may also be called for entry when a screen is exposed by virtue of a window overlaying it being closed or deselected, and called for exit when a window is popped up or selected over the screen in question. This is not the default behavior because it would introduce incompatibilities with earlier releases of JAM.

If you are not concerned with compatibility with earlier releases, it is strongly suggested that you make the following library function call near the beginning of your application, enabling the calling of screen functions when screens are exposed or hidden:

```
xsm_option(EXPHIDE_OPTION, ON_EXPHIDE)
```

Screen functions specified for screen entry via the Screen Editor are invoked after any installed default screen function. Screen functions specified for screen exit via the Screen Editor are called before any installed default screen function.

Screen Function Arguments

All screen functions receive two arguments:

1. The screen name.
2. An integer bitmask containing contextual information about the circumstances under which the function was called.

The contextual information in the second parameter includes the following bit masks:

- `K_ENTRY`
  If this is set (i.e. if the 'bitwise and' of `param4` and `K_ENTRY` is not zero), the function was called on screen entry.
- `K_EXIT`
  If this is set (i.e. if the 'bitwise and' of `param4` and `K_EXIT` is not zero), the function was called on screen exit.
- `K_EXPOSE`
  If this is set (i.e. if the 'bitwise and' of `param4` and `K_EXPOSE` is not zero), the function was called because the screen was selected or deselected, or because a window was popped over the screen or a window that used to be overlaid on the screen was closed[12].
- `K_KEYS`
  Mask for the bits indicating which event caused the screen to be exited. The intersection of this mask and the second parameter to the screen function should be tested for equality against one of the two remaining values below:
- `K_NORMAL`
  If set (i.e. if the 'bitwise and' of `param4` and `K_KEYS` equals `K_NORMAL`), a "normal" call to `xsm_close_window` caused the screen to close.
- `K_OTHER`
  If set (i.e. if the 'bitwise and' of `param4` and `K_KEYS` equals `K_OTHER`), the screen is being closed because another form is being displayed or because `xsm_resetcrt` is called.

Screen Function Return Codes

All screen functions should return 0.

2.2.3.

Control Functions

Control functions are called by the JAM Executive in the processing of control strings and by JPL routines that call PL/1 functions. The JAM Executive will call control functions, if specified and installed, when control strings that start with a caret (^) are executed. JPL procedures may also execute control functions by using the `call` verb.

12. If both `K_ENTRY` and `K_EXPOSE` are set, the screen is being uncovered and activated by virtue of an overlaid window being closed. If both `K_EXIT` and `K_EXPOSE` are set, the screen is being covered and deactivated by virtue of a window being popped up over it.

There is no default control function. Control functions are installed via the `-retain_all` argument to the `bind` command. JPL procedures may be directly specified as control functions by preceding the name of the procedure in a control string with the string "`jpl `".

A number of control functions of general use are built in to JAM. These built–ins can be used by any JAM application. They are listed in Chapter 4.

Control Function Invocation

Control functions are called by the JAM Executive when a control string starting with a caret is processed. Such control strings are often attached, via the Screen Editor, to function keys or to menu selections in control fields. In addition, the JPL verb `call` can be used to invoke control functions.[13]

Control Function Arguments

Control functions receive a single argument, namely a buffer containing a copy of the control string that invoked the function, without the leading caret. It is only the first word on the control string that identifies the function, the rest of the string may contain arbitrary data that can be parsed and used as arguments.

Control Function Return Codes

Control functions may return any integer. The return value from a control function may be used for conditional control branching in target lists (see the Authoring Guide). If there is no target list, and the control string returns a function key which has an associated control string in it's own right, then that control string is executed.

2.2.4.

Key Change Functions

The key change function is called by the Screen Manager as keys are read from the keyboard from within the library routine `xsm_getkey`, which is called in the input processing for all keys by JAM. Only one individual keychange function may be installed at a time.

Keys placed on the queue with the library routine `xsm_ungetkey` or with the built–in control function `^jm-keys` are not processed by the installed key change function.

13 The JPL call verb does not execute control strings. It looks for functions to call.

The key change function is installed as KEYCHG_FUNC via `xsm_n_uinstall`.

Key Change Function Invocation

The key change function is called exactly once for every key read in from the keyboard or supplied by the playback hook function described in section 2.2.10..

Key Change Function Arguments

The key change function is passed a single integer argument, namely the JAM logical key that was read from the keyboard or received from the playback hook function.

Key Change Function Return Codes

The key change function returns the key to be substituted for the one passed as an argument. Any key returned to `xsm_getkey` will be returned by `xsm_getkey` to its caller. However, if the key change function returns 0, `xsm_getkey` will get the next key from the keyboard[14].

2.2.5.

Group Functions

The Screen Manager will call group functions, if specified, on entry, exit, and validation of radio buttons and checklists. Calls to group entry and group exit functions are guaranteed to be paired for each group.

A single default group function may be installed. It will be invoked on entry, exit, and validation for every group. The default group function is installed as via `xsm_n_uinstall`. Group functions specified as entry, exit, or validation functions for a given group in the Screen Editor are installed via the `-retain_all` argument to the `bind` command. JPL procedures may also be directly specified as group functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl groupfunc`.

Please note that field validation functions for fields that are members of groups or menus are called at selection and, in the case of checklists, deselection as discussed above in section 2.2.1. on page 11.

Group Function Invocation

Group functions are called for group entry whenever the cursor enters a group, including the times when the group containing the cursor is activated by virtue of an overlying win-

14. See the library routine `XSM_KEYOPTION` for a different method of changing the function of a logical key.

dow being closed. Group functions are called for group exit whenever the cursor leaves a group, including the times when the group is left because a window is popped up over the existing screen. Group functions are called for validation whenever the group is validated. This occurs at any of the following times:

- As part of group validation, when you exit the group by hitting TAB or making a selection from an autotab group. The BACKTAB and arrow keys do not normally cause validation.
- As part of screen validation when the XMIT key is struck.
- When the application code calls library routines for group validation.

Group functions specified for group entry via the Screen Editor are invoked after any installed default group function. Group functions specified for group exit or validation via the Screen Editor are called before any installed default group function.

Group Function Arguments

All group functions receive two arguments:

1. The group name.
2. An integer containing contextual information about the validation state of the group and the circumstances under which the function was called.

The information contained in the third argument to group functions is identical to that passed in the fourth argument to field functions. See section 2.2.1. on page 11 for an explanation.

Group functions are called for validation regardless of whether the group was previously validated. They may test the `VALIDED` and `MDT` bits to avoid redundant processing.

Group Function Return Codes

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. Any non–zero return code should indicate that the group does not pass validation. If the returned value is 1, the cursor will not be repositioned to the offending group. Any other non–zero return value will cause the cursor to be repositioned to the group that failed the validation.

2.2.6.

Asynchronous Functions

The installed asynchronous function is called periodically by the Screen Manager while the keyboard input routine waits for user input. It can be used to poll or otherwise manipulate communications resources, or to update the display on the screen.

The asynchronous function is installed individually as ASYNC_FUNC via the library routine xsm_async.

Asynchronous Function Invocation

The asynchronous function is called from the very lowest level of JAM keyboard input. When the asynchronous function is installed, the device driver clock on the terminal input device is set to time out on its character read operation, and if a character is not read in that time interval the asynchronous function is invoked before another character read operation is attempted. The time out interval is specified when the function is installed. The time out is measured in tenths of seconds. The maximum interval is 255 (25.5 seconds).

Asynchronous Function Arguments

The asynchronous function is passed no arguments.

Asynchronous Function Return Codes

The asynchronous function should generally return 0. If it returns –1, it will not be called again until at least one additional character has been read from the keyboard. The asynchronous function may return a key, which will be returned to xsm_getkey and on to the application. If that key is a JAM logical key, no further translation will be done. If the asynchronous function returns a data character, JAM will interpret it as a physical keyboard stroke.

2.2.7.

Insert Toggle Functions

The Screen Manager will call the Insert Toggle Function when switching between input and overstrike mode for data entry. Generally this hook function will be used to update some aspect of the display informing the user of the current mode.

The insert toggle function is installed individually as INSCRSR_FUNC via xsm_n_uinstall. JAM automatically installs an insert toggle function that changes the cursor style when the mode is changed. If an application installs its own insert toggle function, the JAM function will be de–installed, and the new insert toggle function may want to call the function directly.

Insert Toggle Function Invocation

The function will be invoked by JAM whenever the data entry mode shifts from insert to overstrike mode or from overstrike to insert mode. Most often, this occurs when the end–user strikes the INSERT key.

Insert Toggle Function Arguments

One integer argument is passed to the insert toggle function. It specifies the mode. If its value is 1, JAM is entering insert mode. If it is 0, JAM is entering overstrike mode.

Insert Toggle Function Return Codes

The insert toggle function should return 0.

2.2.8.

Check Digit Functions

The Screen Manager will call the check digit function for any field that is marked for check digit in the Screen Editor. It may be used to implement any desired check-digit algorithm. If there is no check digit function installed in the application, JAM will use the default library function `xsm_ckdigit`. A new check digit function is installed as `CKDIGIT_FUNC` via the library routine `xsm _n_uinstall`.

Check Digit Function Invocation

The check digit function is called by JAM during validation of fields marked for check digit.

Check Digit Function Arguments

The check digit function is passed the following arguments:

1. The integer number of the field undergoing validation.
2. The field contents.
3. The integer occurrence number for the data undergoing validation.
4. The integer modulus as specified in the Screen Editor.
5. The integer minimum number of digits as specified in the Screen Editor.

Check Digit Function Return Codes

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

not marked as validated. It is assumed that the check digit function display its own error messages.

2.2 9.

Initialization and Reset Functions

The initialization and reset functions are called by the Screen Manager on display setup and display reset respectively. The initialization function can be used to set the terminal type and the reset function can be used to handle any cleanup that the application needs to do whether it is terminated gracefully or not.

Initialization and reset functions are installed individually as `UINIT_FUNC` and `URESET_FUNC` respectively via calls to `xsm_n_uinstall`.

Initialization and Reset Function Invocation

The initialization function is called from the library routine `xsm_initcrt`. When it is called, JAM has not yet allocated its required memory structures, and the physical display characteristics are still untouched by JAM. In general, it is suggested that hook functions be installed after initialization with `xsm_initcrt`, but clearly this is an exception. The initialization function must be installed before `xsm_initcrt` is called. This function is installed as `UINIT_FUNC` via the library routine `xsm_n_uinstall`.

The reset function is called from the library routine `xsm_resetcrt` after JAM has released its memory and reset the physical display characteristics. Since the JAM abort routine `xsm_cancel` calls `xsm_resetcrt` before the application terminates, the reset function is generally called at application exit whether the exit is graceful or not[15]. This function is installed as `URESET_FUNC` via the library routine `xsm_n_uinstall`.

Initialization and Reset Function Arguments

The initialization function is passed a single argument, namely a 30 byte character buffer into which it may place the null-terminated string mnemonic identifying the terminal type in use. This is primarily of use on operating systems without an environment. This function can be used to obtain the terminal type in some system-specific way.

The reset function is passed no arguments.

Initialization and Reset Function Return Codes

Both the initialization and reset hook functions should return 0.

15 Interrupt handlers may need to be set by the developer to insure that `XSM_CANCEL` is called at all the necessary hardware and software interrupt signals. It is suggested that this setup be done in the function installed as an initialization function.

2.2.10.

Recording and Playing Back Keystrokes

The Screen Manager provides hooks for recording and playing back keystrokes. This facility could be used to implement simple macro capabilities, or to perform regression testing on a JAM application. The developer should ensure that the record and playback functions are not in use simultaneously.

Record and playback functions are installed individually as `RECORD_FUNC` and `PLAY_FUNC` respectively via `xsm_n_uinstall`.

Record/Playback Function Invocation

The record function is called from `xsm_getkey` when it has a translated key value in hand that it is about to return to the application. The playback function is called from `xsm_getkey`, when installed, in place of a read from the keyboard[16]. For accurate regression testing, the playback function may need to pause and flush the output to simulate a realistic rate of typing, and may need to call the asynchronous function, if there is one.

Record/Playback Function Arguments

The record function is passed a single integer, which is the JAM logical key to record. Generally that key is recorded in some fashion for a possible playback at a later date. The playback function receives no arguments.

Record/Playback Function Return Codes

The record function should return 0. The playback function should return the logical key that was recorded at an earlier time.

2.2.11.

Status Line Functions

The status line function is called by the Screen Manager whenever the status line is about to be flushed, or physically written to the terminal device. It is intended for use on terminals that require unusual status line processing, beyond the scope of the generic code, but other uses are possible.

16. Since characters are recorded after processing by the key change function but played back before key change translation, some key change functions may interfere with the accurate playback of recorded keystrokes. See the description of `XSM_GETKEY` in the Programmer's Reference Manual for more information.

The status line function is installed individually as `STAT_FUNC` via `xsm_n_uinstall`.

Status Line Function Invocation

The status line function is called when the status line is about to be physically written to the terminal display. Because of delayed write, this may or may not be at the time when the functions that specify message line text are actually called.

Status Line Function Arguments

The status line function receives no arguments. It can access copies of the text and attributes about to be flushed to the status line using the following library routine calls:

```
stat_text = xsm_pinquire(SP_STATLINE);
stat_attr = xsm_pinquire(SP_STATATTR);
```

Status Line Function Return Codes

If the status line function returns 0, JAM continues its usual processing and actually writes out the status line. If the function returns any other value, JAM assumes that the physical write of the status line was handled in the hook function.

2.2.12.

Video Processing Functions

The Screen Manager calls the developer–installed video processing function to allow for special handling of various video sequences by the application. This is a specialized hook required only when the JAM video file is unable to provide support for a particular type of terminal.

The video processing function is installed individually as `VPROC_FUNC` via `xsm_n_uinstall`.

Video Processing Function Invocation

The video processing function is called by JAM's output routine just before a video output operation is about to begin.

Video Processing Function Arguments

The video processing function receives two arguments. The first is an integer video processing code defined in the header file `smvideo.incl.pll` and outlined in the table

below. The second is an array of integers with parameters for the video processing code. The number of parameters passed depends on the operation as shown in the table below. For video processing codes that require no arguments, a NULL is passed.

Code	*Operation Description*	*# of params*
V_ARGR	remove area attribute	
V_ASGR	set area graphics rendition	11
V_BELL	visible alarm sequence	
V_CMSG	close message line	
V_COF	turn cursor off	
V_CON	turn cursor on	
V_CUB	cursor back (left)	1
V_CUD	cursor down	1
V_CUF	cursor forward (right)	1
V_CUP	set cursor position (absolute)	2
V_CUU	cursor up	1
V_ED	erase entire display	
V_EL	erase to end of line	
V_EW	erase window to background	5
V_INIT	initialization string	
V_INSON	set insert cursor style	
V_INSOFF	set overstrike cursor style	
V_KSET	write to soft key label	2
V_MODE4	single character graphics mode (also V_MODE5, 6)	
V_MODE0	set graphics mode (also V_MODE1, 2, 3)	
V_OMSG	open message line	

Code	*Operation Description*	*# of params*
V_RESET	reset string	
V_RCP	restore cursor position	
V_REPT	repeat character sequence	2
V_SCP	save cursor position	
V_SGR	set latch graphics rendition	11

Video Processing Function Return Codes

When the video processing function returns 0, JAM will continue with normal processing. If it returns any other value, JAM will assume that the operation has been handled in the hook function. This allows the developer to implement only necessary operations.

Other Hook Functions

The Screen Manager provides an additional hook to handle block mode terminals. This function is best viewed as a driver. Block mode is described in Chapter 10.

2.3. CODING STRATEGY, RULES AND PITFALLS

2.3.1.
Displaying Screens

There are a number of library functions provided for the display of screens as forms or windows. In general, the following rules and guidelines should be followed in choosing between them and deciding when they can be used:

- The display of screens as forms or windows from within screen functions at screen entry or screen exit is neither recommended nor supported.

- The routines `xsm_jform`, `xsm_jwindow`, and `xsm_jclose` are provided specifically for the display and destruction of screens in applications that use the JAM Executive. Applications not using the JAM Executive should not use these routines. They are recommended over the other screen display routines in applications that do use the JAM Executive.
- The form display routine `xsm_jform` manipulates the form stack appropriately. The use of any other form display routines in applications that use the JAM Executive will exhibit unexpected behavior, as the form stack will not be synchronized with the application flow.

2 3.2.

Recursion

The developer should be careful, when using hook functions, to avoid the recursion that will come from nested hook function calls. Such recursion will not be easy to detect in the source code itself: some understanding of the product mechanism is required.

For example, care should be taken when writing record, playback, or key change functions that read from the keyboard, or status line functions that themselves cause the status line to be flushed. A default screen entry function that in and of itself opens new screens could be a problem.

Chapter 3.
Local Data Block

The Local Data Block, or LDB, is a region of memory for the storage of JAM field data that is generally shared between screens. It is discussed in the JAM Development Overview and in the Author's Guide.

3.1.
LDB CREATION

The LDB is created with the library routine call `xsm_ldb_init`. This routine searches for a data dictionary file created from the authoring tool with the Data Dictionary Editor. For more information about the data dictionary and the Data Dictionary Editor, see the Author's Guide.

If the data dictionary file is found, it is read and a single LDB entry is created in memory for every data dictionary entry that has a non–zero scope. Note that only the name of the LDB entry is placed in memory, storage for the field data that is stored with the entry is not allocated until the entry is used.

After it is created, the LDB is initialized from ASCII text files. These files, described in the Author's Guide, contain pairs of LDB names and values. The LDB entries named are filled with the values that follow them in the files.

3.2.
HOW JAM USES THE LDB

JAM uses the LDB for the storage and propagation of field data from screen to screen in the application. Every time a screen is opened, or exposed by the closing of a window that

covers it, every field on the screen named identically to an LDB entry is filled with the value of the LDB entry. This occurs after the screen entry function is called.

Correspondingly, every time a screen is closed, or hidden when a window pops up over it, every LDB entry that is named identically to a field on the screen is filled with the value of the screen field. This occurs before the screen exit function is called.

When a screen is populated from the LDB at screen entry time, there is a subtle difference between a new screen being opened and a screen being exposed when a covering window is closed. When a screen is newly opened, only empty fields with corresponding LDB entries will be populated from the LDB. When a screen is exposed, all fields that have corresponding LDB entries will be populated.

3.3. LDB ACCESS

Data in the LDB can be accessed with the library routines `xsm_n_getfield`, `xsm_n_putfield`, `xsm_i_getfield`, `xsm_i_putfield`, and related functions that access data by field name. These routines access the data on the current screen if the field that is named exists on the current screen. If the field does not exist on the current screen, these routines access the LDB.

During screen entry and exit processing only, the search order is reversed. During the screen entry and exit functions, these access routines first search the LDB and then search the screen. This is because the LDB is merged to the screen after the screen entry function, and the screen is stored to the LDB before the screen exit function. If the search order were not reversed the data accessed would be invalid[18].

18 This could, in a very small number of cases, introduce some incompatibilities with applications that were written with earlier releases of JAM. If such compatibility problems arise, use the library function `XSM_OPTION` setting the option `ENTEXT_OPTION` to `FORM_FIRST`.

Chapter 4.
Built–in Control Functions

This section describes control functions supplied with JAM. Note that the synopsis is for a JAM control string, not a programming language source statement. The return value of a control function can be used in a target list; see the Author's Guide for information on control strings and target lists.

You may use these functions in control strings and in JPL `call` statements.

jm_exit

end processing and leave the current screen

SYNOPSIS

```
^jm_exit
```

DESCRIPTION

Clears the current form or window and returns to the previous one. If the current form is the application's top–level form, JAM will prompt and exit to the operating system.

The effect is like the default action of the run–time system's EXIT key.

EXAMPLE

The following control string invokes a function named `process`. If it returns 0, another function is invoked to reinitialize the screen; but if it returns –1, the screen is exited. See `jm_gotop` for another example.

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

The example below shows how a form or a window can be replaced by another form or a window:

```
^(0=&w2)jm_exit
```

jm_gotop

return to application's top–level form

SYNOPSIS

```
^jm_gotop
```

DESCRIPTION

Returns to the application's top–level screen, ordinarily the first screen to appear when the application was run. All forms on the form stack and windows on the window stack are discarded.

The run–time system's SPF1 key performs the same action, unless you change it using `SMINICTRL`.

EXAMPLE

The following menu makes use of both `jm_exit` and `jm_gotop`.

```
+-------------------------------------------------+
:                                                 :
:  Query customer database__    custquery.jam__   :
:  Update customer database_    custupdate.jam_   :
:  Free-form query_________     'sql___________   :
:  Return to previous menu__    ^jm_exit_______   :
:  Return to main menu______    ^jm_gotop______   :
:                                                 :
+-------------------------------------------------+
```

jm_goform

prompt for and display an arbitrary form

SYNOPSIS

```
^jm_goform
```

DESCRIPTION

This function pops up a window in which you may enter the name of a form; it will then close all open windows and attempt to display the form, as if that form's name had appeared in a control string. It is useful for providing a shortcut around a menu system for experienced users.

The result is the same as the default action of the run–time system's SPF3 key.

EXAMPLE

The following line, if placed in your setup file, will make the PF10 key act like SPF3 normally does:

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

jm_keys
simulate keyboard input

SYNOPSIS

```
^jm_keys keyname-or-string {keyname-or-string ...}
```

DESCRIPTION

Queues characters and function keys that appear after the function name for input to the run–time system, using `xsm_ungetkey`. The run–time system then behaves as though you had typed the keys.

Function keys should be written using the logical key mnemonics listed in *smkeys.incl.pl1* . Data characters should be enclosed between apostrophes `''`, back-quotes `` `` ``, or double quotes `""`. This function passes its arguments to `xsm_ungetkey` in reverse order, so you supply them in the natural order.

`jm_keys` will process a maximum of 20 keys. This limit includes function keys plus characters contained in strings.

EXAMPLE

Enter the name of your favorite bar, followed by a tab and the name of its owner:

```
^jm_keys 'Steinway Brauhall' TAB "James O'Shaughnessy"
```

Return to the preceding menu and choose the second option:

```
^jm_keys EXIT HOME TAB XMIT
```

jm_mnutogl

switch between menu and data entry mode on a dual–purpose screen

SYNOPSIS

```
^jm_mnutogl {screen-mode}
```

DESCRIPTION

JAM supports the use of a single screen for both menu selection and data entry; one popular example is a data entry screen with a "menu bar". The screen must, however, be either one or the other at any given moment. This function switches the run–time system's treatment of the screen to the other mode. This function performs the same function as the MTGL logical key.

An optional argument may be specified which will force the screen into a particular mode, regardless of its current state. To specify menu mode, use the argument 'M' (or 'm'). To specify open–keyboard (data entry) mode, use the argument 'O' (or 'o').

jm_system

prompt for and execute an operating system command

SYNOPSIS

```
^jm_system
```

DESCRIPTION

This function pops up a small window, in which you may enter an operating system command. When you press TRANSMIT, it closes the window and executes the command. While the command is executing, your terminal is returned to the operating system's default I/O mode.

The run–time system's SPF2 key invokes this function by default.

EXAMPLE

The following line, when placed in your setup file, will cause the PF10 key to act as SPF2 normally does:

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

jm_winsize

allow end–user to interactively move and resize a window

SYNOPSIS

```
^jm_winsize
```

DESCRIPTION

Calling `jm_winsize` has the same effect as if the end–user had just hit the VWPT (viewport) logical key. The viewport status line appears and the user can move, resize and change the offset of the screen as well as move to any sibling windows. When the end–user hits XMIT (transmit) the previous status line is restored.

In order for the end–user to able to move from one window to another, the windows must be siblings. Windows may be specified as siblings by specifying `&&` in a JAM control string. See the sections on "Viewports and Positioning" and "Control Strings" in the Author's Guide for further information. This function parallels the library routine `xsm_winsize`.

jpl

invoke a JPL procedure

SYNOPSIS

```
^jpl procedure [ argument ... ]
```

DESCRIPTION

This function invokes a procedure written in the JYACC Procedural Language. procedure should be the name of a JPL procedure or module; anything following that will be passed to the procedure as arguments. See the JPL Programmer's Guide for the rules used by the JPL interpreter to determine which JPL procedure is executed. The value returned by your procedure will be returned by `jpl` for use in a target list.

This function is similar to the JPL `jpl` command. Colon expansion is done on the arguments.

EXAMPLE

The control string below invokes a JPL function to concatenate two strings and store the result in `target`.

```
^jpl concat target "king" "kong"
```

Chapter 5.
Keyboard Input

Keystrokes are processed in three steps. First, the sequence of characters generated by one key is identified. Next the sequence is translated to an internal value, or logical character. Finally, the internal value is either acted upon or returned to the application ("key routing"). All three steps are table–driven. Hooks are provided at several points for application processing; they are described in the chapter "Writing and Installing Hook Functions".

5.1.
LOGICAL KEYS

JAM processes characters internally as logical values, which frequently (but not always) correspond to the physical ASCII codes used by terminal keyboards and displays. Specific physical keys or sequences of physical keys are mapped to logical values by the key translation table, and logical characters are mapped to video output by the MODE and GRAPH commands in the video file. For most keys, such as the normal displayable characters, no explicit mapping is necessary. Certain ranges of logical characters are interpreted specially by JAM; they are

- `0x0100 to 0x01ff`: operations such as tab, scrolling, cursor motion
- `0x6101 to 0x7801`: function keys PF1 – PF24
- `0x4101 to 0x5801`: shifted function keys SPF1 – SPF24
- `0x6102 to 0x7802`: application keys APP1 – APP24

5.2.
KEY TRANSLATION

The first two steps together are controlled by the key translation table, which is loaded during initialization. The name of the table is found in the environment (see the configuration guide for details). The table itself is derived from an ASCII file which can be modified by any editor; a screen-oriented utility, `modkey`, is also supplied for creating and modifying key translation tables (see the Utilities Guide).

JAM assumes that the first character of any multi-character key sequence to be translated to a single logical key is a control character in the ASCII chart (0x00 to 0x1f, 0x7f, 0x80 to 0x9f, or 0xff). All characters not in this range are assumed to be displayable characters and are not translated.

Upon receipt of a control character, the keyboard input function `xsm_getkey` searches the translation table. If no match is found on the first character, the key is accepted without translation. If a full match is found on the first character, an exact match has been found, and `xsm_getkey` returns the value indicated in the table. The search continues through subsequent characters until either

1. an exact match on n characters is found and the n+1'th character in the table is zero, or n is 6. In this case the value in the table is returned.
2. an exact match is found on n–1 characters but not on n. In this case `xsm_getkey` attempts to flush the sequence of characters returned by the key.

This last step is of some importance: if the operator presses a function key that is not in the table, the Screen Manager must know "where the key ends". The algorithm used is as follows. The table 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 no entry matches by type at the n'th character, the shortest sequence that matches on n–1 characters is used.) This method allows `xsm_getkey` to distinguish, for example, between the sequences `ESC O x`, `ESC [ A`, and `ESC [ 1 0 ~`.

5.3.
KEY ROUTING

The main routine for keyboard processing is `xsm_input`. This routine calls `xsm_getkey` to obtain the translated value of the key. It then decides what to do based on the following rules.

If the value is greater than 0x1ff, `xsm_input` returns to the caller with this value as the return code.

If the value is between 0x01 and 0x1ff, the key is first translated via the key translation table. This table is changed with the library routine `xsm_keyoption`. Then processing is determined by a routing table. Use `xsm_keyoption` to get and set the routing information for a particular key. The routing value consists of two bits, examined independently, so four different actions are possible:

1. If neither bit is set, the key is ignored.
2. If the EXECUTE bit is set and the 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.
3. If the RETURN bit is set, `xsm_input` returns the logical value to the caller; otherwise, `xsm_getkey` is called for another value.
4. If both bits are set, the key is executed and then returned.

The default settings are *ignore* 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 exteneded ASCII characters are ignored. The application function keys (PF1–24, SPF1–24, APP1–24, 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.

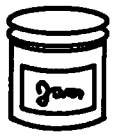
Applications can change key actions on the fly by using `xsm_keyoption`. For example, to disable the backtab key the application program would execute

```
call xsm_keyoption(BACK, KEY_ROUTING, KEY_IGNORE)
```

To make the field erase key return to the application program, use

```
call xsm_keyoption(FERA, KEY_ROUTING, RETURN)
```

Key mnemonics can be found in the file `smkeys.incl.pll`.



Chapter 6.
Terminal Output Processing

JAM uses a sophisticated *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 being solicited or the library function `xsm_flush` has been called. Instead, the run–time system does screen updates in memory, and keeps track of the display positions thus "dirtied". Flushing begins when the keyboard is opened; but if you type a character while flushing is incomplete, the run–time system will process it before sending any more output to the display. This makes it possible to type ahead on slow lines. You may force the display to be updated by calling `xsm_flush`.

JAM takes pains to avoid code specific to particular displays or terminals. To achieve this it defines a set of logical screen operations (such as "position the cursor"), and stores the character sequences for performing these operations on each type of display in a file specific to the display. Logical display operations and the coding of sequences are detailed in the Video Manual; the following sections describe additional ways in which applications may use the information encoded in the video file.

6.1. GRAPHICS CHARACTERS AND ALTERNATE CHARACTER SETS

Many terminals support the display of graphics or special characters through alternate character sets. Control sequences switch the terminal among the various sets, and characters in the standard ASCII range are displayed differently in different sets. JAM supports alternate character sets via the `MODEx` and `GRAPH` commands 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 will first transmit the sequence that switches to `mode`, then transmit `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.

6.2. THE STATUS LINE

JAM reserves one line on the display for error and other status messages. Many terminals have a special status line (not addressable with normal cursor positioning); if such is not the case, JAM will use the bottom line of the display for messages. There are several sorts of messages that use the status line; they appear below in priority order.

1. Transient messages issued by `xsm_err_reset` or a related function
2. Ready/wait status
3. Messages installed with `xsm_d_msg_line` or `xsm_msg`
4. Field status text
5. Background status text

There are several routines that display a message on the status line, wait for acknowledgement from the operator, and then reset the status line to its previous state: `xsm_query_msg`, `xsm_err_reset`, `xsm_emsg`, `xsm_quiet_err`, and `xsm_qui_msg`. `xsm_query_msg` waits for a yes/no response, which it returns to the calling program; the others wait for you to acknowledge the message. These messages have highest precedence.

`xsm_setstatus` provides an alternating pair of background messages, which have next highest precedence. Whenever the keyboard is open for input the status line displays `Ready`; it displays `Wait` when your program is processing and the keyboard is not open. The strings may be altered by changing the `SM_READY` and `SM_WAIT` entries in the message file.

If you call `xsm_d_msg_line`, the display attribute and message text you pass remain on the status line until erased by another call or overridden by a message of higher precedence.

When the status line has no higher priority text, the Screen Manager checks the current field for text to be displayed on the status line. If the cursor is not in a field, or if it is in a field with no status text, JAM looks for background status text, the lowest priority. Background status text can be set by calling `xsm_setbkstat`, passing it the message text and display attribute.

In addition to messages, the rightmost part of the status line can display the cursor's current screen position, as, for example, `C 2,18`. This display is controlled by calls to `xsm_c_vis`.

During debugging, calls to `xsm_err_reset` or `xsm_quiet_err` can be used to provide status information to the programmer without disturbing the main screen display. Keep in mind that these calls will work properly only after screen handling has been initialized by a call to `xsm_initcrt`. `xsm_err_reset` and `xsm_quiet_err` can be called with a message text that is defined locally, as in:

```
call xsm_err_reset("ZIP CODE INVALID FOR THIS STATE.");
```

However, the JAM library functions use a set of messages defined in an internal message table. This table is accessed by the function `xsm_msg_get`, using a set of defines in the header file `smerror.incl.pl1`. The return value from `xsm_msg_get` can be used as input for one of the status line functions.

The message table is initialized from the message file identified by the environment variable SMMSGS. Application messages can also be placed in the message file. See the section on message files in the Configuration Guide.

Chapter 7.
Writing International (8 bit) Applications

7.1. INTRODUCTION

This chapter describes how to use the 8 bit internationalization capabilities that have been incorporated into JAM Release 5.

From the point of view of someone who has used JAM without these features, a few differences will be apparent immediately. Other, more subtle, differences will emerge as the package is used in building language–independent applications. Finally, many of the changes were made so that the development utilities could be localized for use in other countries. These will largly go unnoticed by people using the package in English.

7.1.1. General Overview

The purpose of the 8 bit NLS is to allow the JAM product and applications created with with it to be "localized" for use in non–English–speaking countries. This means that the product can be made to look like it originated in 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. Notwithstanding this, many of the features that are only visible to programmers will continue to be in English since many programmers are used to working in English.

The capabilities described are limited to languages in which characters can be represented in 8 bits of information and those that use a left–to–right entry order. This eliminates the complexities associated with many far– and middle–eastern languages.

7.2. LOCALIZATION

JAM and JAM applications can be localized by taking the following steps:

- Use the Screen Editor to translate all screens in the application.
- Modify and recompile the message file.
- Translate the documentation.

7.2.1. Background

The JAM product was originally developed with some internationalization issues in mind. It has always used 8 bit character data, without appropriating a bit for internal use. So one of the major demands of the international market was already satisfied.

Date and time formats have always been completely specified by the screen creator. The wide variety of formats available in Release 4 could satisfy most requirements. In Release 5, additional capabilities were added to make it easier to convert screens from one language to another. Currency formats were the least international of the features in the Release 4 product. Release 5 makes these completely language independent.

Each of the sections below discusses some aspect of internationalization.

7.2.2. 8 Bit Character Data

As pointed out in the introduction, JAM supports 8 bit character data. Video files specific to the terminal can give special instructions, if necessary, as to 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 will not need to shift character sets.

The video file can also be used to translate between two different standards for international characters. Thus the screens could 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 character set (e.g. 0x01 – 0x1f, or 0x80 – 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. The cursor control keys (arrows, tab, etc.), function keys, and soft 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 will 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, `modkey` (or a text editor) would be used to map the byte sequences into a logical value, just as the video file would be used to map the logical value to the sequence required by the display terminal.

If you have questions about how to display non–English characters or to receive them from the keyboard, consult the chapters on keyboard and video processing.

7.2.3.
Date And Time Fields

Date and Time fields have been completely revamped in Release 5. They have been combined to enable one field to have both date and time information. This, and the fact that more flexibility was added to date and time formatting, required changes to the date and time mnemonics. For example, in Release 4, the mnemonic `mm` was used for a 2–digit month in Date fields as well as the specifier for minutes in Time fields. Clearly, this cannot serve both purposes when the fields are combined.

In Release 5, the mnemonics for specifying date and time formats are stored in the message file so they may be changed. In addition, they are stored in a "tokenized" form internally which provides two major benefits. First, the need to parse the formats at runtime is eliminated, thus speeding up processing and reducing memory requirements. Second, screen designers in different countries editing the same screen will all see date and time specifications in formats they are used to. For example, if an English screen designer created a date field with the format `mon/day/year`, it might show up on a French system as `mois/jour/annee`.

The problem of interchanging the month and day is dealt with later.

The table below shows the default message file entries for date and time mnemonics:

Msg # Mnemonic	*Date/Time Mnemonic*	*Tokenized Format*	*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	abbrev. month name
FM_MONL	MONL	%*m	long month name

Thus, a date field specified as mm/dd/yyyy in Release 4 would be MON2/DATE2/YR4 in Release 5. The f4to5 conversion program will convert the format to %m/%d/%4y internally so it will automatically show up correctly when the screen is edited. The mnemonics were chosen to correspond to ANSI standards. You can change them to suit your own needs by simply changing the message file and running msg2bin. To change the mnemonic for a 4 digit year from YR4 to YYYY, for example, change the message file line

```
FM_YR4 = YR4
```

to

```
FM_YR4 = YYYY
```

and run `msg2bin`.

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. What is important, however, is to be able to modify an application to operate in a different language. The goal is that only the text of the screens and the message file should need to be changed.

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

```
Mon Apr 4, 1989
```

whereas on a French system it would be

```
Lun Avr 4, 1989
```

This happens without changing the date format. All that has changed are the names and abbreviations of the months and days which are also stored in the message file so it is a simple matter to convert them.

Now consider a date field which in English should show up in `mm/dd/yyyy` form but should appear in French as `dd-mm-yyyy`. In this case, the date format itself would have to be modified. For this reason, 10 additional formats are supplied for the designer's use. For instance, in the message file the designer can specify a new date mnemonic called `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 screen, needs to be changed to convert the date field to French.

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.

The following table shows the default message file entries for these extra date mnemonics:

Msg Number Mnemonic	Date/Time Mnemonic	Tokenized Form	Corresponding Msgfile Entry	Default
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

Msg Number Mnemonic	Date/Time Mnemonic	Tokenized Form	Corresponding Msgfile Entry	Default
FM_2MN_DEF_DT	DEFAULT TIME	%2f	SM_2DEF_DTIME	%h:%0M
FM_3MN_DEF_DT	DEFAULT3	%3f	SM_3DEF_DTIME	%m/%d/%2y %h:%0M
FM_4MN_DEF_DT	DEFAULT4	%4f	SM_4DEF_DTIME	%m/%d/%2y %h:%0M
FM_5MN_DEF_DT	DEFAULT5	%5f	SM_5DEF_DTIME	%m/%d/%2y %h:%0M
FM_6MN_DEF_DT	DEFAULT6	%6f	SM_6DEF_DTIME	%m/%d/%2y %h.%0M
FM_7MN_DEF_DT	DEFAULT7	%7f	SM_7DEF_DTIME	%m/%d/%2y %h:%0M
FM_8MN_DEF_DT	DEFAULT8	%8f	SM_8DEF_DTIME	%m/%d/%2y %h.%0M
FM_9MN_DEF_DT	DEFAULT9	%9f	SM_9DEF_DTIME	%m/%d/%2y %h:%0M

Thus, if the screen designer specifies a date field with the format `DEFAULT DATE`, it would show up in `mm/dd/yy` form. If the line

```
SM_1DEF_DTIME = %m/%d/%2y
```

in the message file were changed to

```
SM_1DEF_DTIME = %d-%m-%2y
```

the date would show up in `dd-mm-yy` form. To change the mnemonic for this date format to `REGULAR DATE`, the message `FM_1MN_DEF_DT` should be modified.

7.2.4.
Currency Fields

Like Date and Time fields, Currency fields have been modified in Release 5. Since it is not uncommon in Europe to be dealing with several currencies simultaneously, release 5

does not force any one system on the screen creator. Thus, the formatting capabilities were enhanced to support any convention the screen creator might desire. As with date and time formats, a "default" format is supplied that causes the actual format to be taken from the message file. For Currency fields however, this option is supplied only for the parts of the format that may vary from one currency to another.

The new release allows the following items to be specified for Currency fields:

- the decimal symbol (usually dot or comma)
- minimum number of decimal places
- maximum number of decimal places
- thousands separator (usually dot or comma; b = blank)
- the currency symbol to be used (up to 5 characters)
- the placement of that symbol (left, right or at decimal pt)
- default currency from the message file (to replace the above entries)
- rounding (round–up, round–down, round–adjust)
- fill character
- justification
- clear if zero
- apply if empty

There is a slight problem in specifying currency symbols when using the Screen Editor. Since the currency symbol is entered into a regular field, it is not possible to enter trailing spaces (they are always stripped off). Thus, to specify a leading currency symbol separated from the data by a space (`FF 123.456,78`) you must use the message file. For this reason, the dot (`.`) may be used to signify a space when entered into the currency field. A dot in the message file for this purpose will appear as a dot.

The default currency formats are strings of the form ***rmxtpccccc*** where:

- ***r*** = decimal symbol (usually comma or dot)
- ***m*** = minimum number of decimal places
- ***x*** = maximum number of decimal places
- ***t*** = thousands separator (usually comma or dot; b = blank)
- ***p*** = placement of currency symbol (l, r or m)
- ***ccccc*** = up to 5 characters for the currency symbol

Thus, if the screen designer specifies a currency field with the format CURRENCY, it would show up in `$999,999.99` form. If the line

```
SM_0DEF_CURR = ".22,1$"
```

in the message file were changed to

```
SM_0DEF_CURR = ",22.1FF"
```

the field would show up as `FF 999.99,99`. To change the mnemonic for this currency field, the message `FM_0MN_CURRDEF` should be modified. The following table shows the default message file entries for the currency mnemonics:

Msg Number Mnemonic	*Currency Mnemonic*	*Corresponding Msgfile Entry*	*Default*
FM_0MN_CURRDEF	CURRENCY	SM_0DEF_CURR	.22,1$
FM_1MN_CURRDEF	NUMERIC	SM_1DEF_CURR	.09,
FM_2MN_CURRDEF	PLAIN	SM_2DEF_CURR	.09
FM_3MN_CURRDEF	DEFAULT3	SM_3DEF_CURR	.09
FM_4MN_CURRDEF	DEFAULT4	SM_4DEF_CURR	.09
FM_5MN_CURRDEF	DEFAULT5	SM_5DEF_CURR	.09
FM_6MN_CURRDEF	DEFAULT6	SM_6DEF_CURR	.09
FM_7MN_CURRDEF	DEFAULT7	SM_7DEF_CURR	.09
FM_8MN_CURRDEF	DEFAULT8	SM_8DEF_CURR	.09
FM_9MN_CURRDEF	DEFAULT9	SM_9DEF_CURR	.09

7.2.5.
Decimal Symbols

JAM 5 will accomodate 3 decimal symbols which are used in different circumstances:

- System Decimal Symbol
- Local Decimal Symbol
- Field Decimal Symbol

The System Decimal Symbol is the one that library routines like `atof` and `sprintf` use. The Local Decimal Symbol is the one that is used when local customs are followed

(dot in English; comma in French). The Field Decimal Symbol is the one specified for a given field if that field is not observing local conventions.

The System and Local Decimal Symbols are obtained from the operating system if the operating system supports such things (see the installation notes for JAM for your operating system). The Local Decimal Symbol may be specified in the message file (message `SM_DECIMAL`), in which case it overrides the operating system decimal symbol. Dot is the system decimal if no symbol is specified in the message file and if the operating system does not supply one.

The sections below describe the circumstances under which each of the different symbols is used.

7.2.6.
Character Filters

The one time that JAM requires some knowledge of the meaning of the data is while enforcing the character filters on a field. The filters currently supported are digits only, numeric, alphabetic, alphanumeric, and yes/no and regular expression.

To validate the data JAM uses the standard C macros: `isdigit`, `isalpha`, etc. JAM 5 assumes that the operating system supplies these macros in a form suitable for international use. In 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 will supply information about these characters, the proposed ANSI standard does not address the issue. Therefore, for reasons of portability, JAM will continue 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 5 uses the ASCII collating sequence for ranges of characters. Therefore, the expression

```
[a-z]*
```

will match only the English lower case letters. The European character a, for example, would not be matched by this expression.

7.2.7.
Status And Error Messages

All messages produced by JAM 5 are stored in the message file so they may be easily localized. Each message is a complete phrase or sentence. Message components are never pieced together because doing so would make it difficult to translate to a language that has a sentence structure different from English.

7.2.8.
Screens In The Utilities

These screens were memory resident in Release 4. For international customers they must be modifiable.

A linkable `jxform` is be provided, and the library containing the source for the screens is made available. A developer may translate the screens and relink the utilities. Similarly `modkey` is developer–linkable, and the source for its screens is provided. In this way the screens remain memory resident and no compromise of speed need be made.

Unfortunately this solution is not ideal if several users on the same machine wish to use different languages. To support this, the screens may be kept on disk. The current mechanism of `SMPATH` allows run–time selection of the set of screens to be used.

7.2.9.
Screens In Application Programs

The same approach as discussed in the above section can be used for screens in application programs. Thus different language screens can be kept in separate directories and the user can specify which is to be used at run–time.

7.2.10.
Menu Processing

`xsm_input` returns the first character of the selected entry. This, of course, is not language independent. JAM utilities have been modified to use the current field number

rather than the return value. Because it cannot be assumed that all entries will have unique first letters, the `string` option is specified.

Application programs intended for an international market should not rely on the initial character of the menu selection. The field number containing the cursor is a better way of determining which selection the operator has made. However the field numbers may change if the screen is redesigned. Note that this is not a problem when the JAM Executive is used, since the JAM Executive uses relative field numbers to determine the control string to execute when a menu field is selected.

A new additional edit was instituted in JAM 4 that specifies the return code from a `return entry` (or `menu`) field. The screen creator specifies the return code (an integer) when designing the screen. If this edit exists, `xsm_input` uses that value as the return code to the calling program. If this edit does not exist, the usual return code is used.

7.2.11.
`lstform`, `lstdd`, and `jammap`

These utilities list data about the screen in English. Since they are often used for documentation it is important that the text be translatable to other languages. Thus the textual material, headings, etc., have been moved to the message file.

7.2.12.
Range Checks

Range checks for numeric data are presently correctly handled since they use `atof` (assuming that the "strip" routine works properly).

Alphabet data presents special problems. 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 compare equal. Also, in a telephone book, `St.` and `Saint` compare equal, 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 given 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.

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.

For these reasons no changes have been made to the Release 4 method of range checking. `strcmp` and `memcmp` continue to be used. These compare the internal values of the characters, without regard to their meanings in the local language.

7.2.13.
Calculations Using `@SUM` and `@DATE`

These keywords have been retained even though they are language specific. Computations with dates assume the Gregorian calendar. No provison is made for other calendars.

7.2.14.
`xsm_dblval` and `xsm_dtofield`

These routines use `atof` and `sprintf` therefore correctly interpret the System Decimal Symbol (radix character).

7.2.15.
`xsm_is_yes` and `xsm_query_msg`

These routines use the characters in the message file for `y` and `n` and thus are already internationalized. They use `toupper` to recognize the upper case variations.

7.2.16.
Batch Utilities

All the utilities messages, including usage messages have been moved to the message file.

The mnemonics for logical keys (XMIT, EXIT, etc.) are not translated to other languages, nor the mnemonics used in the video file, so the internal processing of the utilities need not be modified.

Chapter 8.
Writing Portable Applications

The following section describes features of hardware and operating system software that can cause JAM to behave in a non–uniform fashion. An application designer wishing to create programs that run across a variety of systems will need to be aware of these factors.

8.1.
TERMINAL DEPENDENCIES

JAM can run on display terminals of any size. On terminals without a separately addressable status line, JAM will steal the bottom line of the display (often the 24th) for a status line, and status messages will overlay whatever is on that line. A good lowest common denominator for screen sizes is 23 lines by 80 columns, including the border (21 if two–line soft key labels will be used).

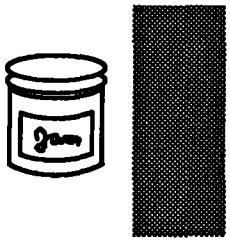
Different terminals support different sets of attributes. JAM makes sensible compromises based on the attributes available; but programs that rely extensively on attribute manipulation to highlight data may be confusing to users of terminals with an insufficient number of attributes. Colors will not show up on monochrome terminals, e.g. Use of graphics character sets is particularly terminal dependent.

Attribute handling can also affect the spacing of fields and text. In particular, anyone designing screens to run on terminals with onscreen attributes must remember to 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).

The key translation table mechanism supports the assignment of any key or key sequence to a particular logical character. However, the number and labelling of function keys on particular keyboards can constrain the application designer who makes heavy use of func-

tion keys for program control. The standard VT100, for instance, has only four function keys. For simple choices among alternatives, menus are probably better than switching on function keys.

Using function key labels, or keytops, instead of hard–coded key names is also important to making an application run smoothly on a variety of terminals. Field status text and other status line messages can have keytops inserted automatically, using the `%K` escape. No such translation is done for strings written to fields; in such cases, you may want to place the strings in a message file, since the setup file can specify terminal–dependent message files.



Chapter 9.
Writing Efficient Applications

9.1.
MEMORY–RESIDENT SCREENS

Memory–resident screens are much quicker to display than disk–resident screens, since no disk access is necessary to obtain the screen data. However, the screens must first be converted to source language modules with `bin2pl1` or a related utility (see the Utilities Guide), then compiled and linked with the application program.

`xsm_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 will generally have the same name as the file the original screen was originally stored in.

A more flexible way of achieving the same object is to use a memory–resident screen list. Bear in mind that the JAM Screen Editor can only operate on disk files, so that altering memory–resident screens during program development requires a tedious cycle of test – edit – reinsert with `bin2pl1` – recompile. The JAM library maintains an internal list of memory–resident screens that `xsm_r_window` and related functions examine. Any screen found in the list will be displayed from memory, while screens not in the list will be sought on disk. This means that the application can be coded to use one set of calls, the r–version, and screens can be configured as disk– or memory–resident simply by altering the list.

Call `xsm_formlist` to add a screen to JAM's memory–resident screen list.

Using memory–resident screens (and configuration files, see the next section) is, of course, a space–time tradeoff: increased memory usage for better speed.

JAM will append the extension found in the setup variable SMFEXTENSION to screen names (*e g.* in control fields) that do not already contain an extension; you must take this into account when creating the screen list. JAM may also convert the name to uppercase before searching the screen list; this is governed by the SMFCASE variable.

9.2. MEMORY–RESIDENT CONFIGURATION FILES

Any or all of the three configuration files required by JAM can be made memory resident. First a PL/1 source file must be created from the binary version of the file, using the `bin2pl1` utility; see the Utilities Guide. The source files created are not readily decipherable. A call is then made to either `xsm_msgread`, `xsm_vinit`, or `xsm_ keyinit`, depending on the type of configuration file being installed.

If a file is made memory–resident, the corresponding environment variable or SMVARS entry can be dispensed with.

9.3. MESSAGE FILE OPTIONS

If you need to conserve memory and have a large number of messages in message files, you can make use of the MSG_DSK option to `xsm_msgread`. This option avoids loading the message files into memory; instead, they are left open, and the messages are fetched from disk when needed. Bear in mind that this uses up additional file descriptors, and that buffering the open file consumes a certain amount of system memory; you will gain little unless your message files are quite large.

9.4. AVOIDING UNNECESSARY SCREEN OUTPUT

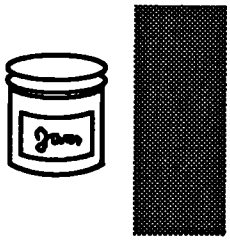
Several of the entries in the JAM video file are not logically necessary, but are there solely to decrease the number of characters transmitted to paint a given screen. This can have a great impact on the response time of applications, especially on time–shared systems

with low data rates; but it is noticeable even at 9600 baud. To take an example: JAM can do all its cursor positioning using the CUP (absolute cursor position) command. However, it will use the 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 is capable of saving and restoring the cursor position itself (SCP, RCP), JAM will use those sequences instead of the more verbose CUP.

The global variable `I_NODISP` may also be used to decrease screen output. While this variable is set to 0 (via `xsm_iset`), calls into the JAM library will cause the internal screen image to be updated, but nothing will be written to the actual display; the display can be brought up to date by resetting `I_NODISP` to 1 and calling `xsm_rescreen`. With the implementation of delayed write this sort of trick is rarely necessary.

9.5. JPL VS. COMPILED LANGUAGES

JPL code execution goes through an extra layer of intrepretation that compiled code, such as PL/1, does not. In most cases, the total run time is too small to matter, but if a JPL function is long or loops many times and a delay is noted, it may pay to rewrite it in PL/1.



Chapter 10.
Block Mode

The purpose of this document is to describe the block mode capabilities of JAM from the perspective of someone using the system and from the perspective of a developer that needs to write a block mode driver.

10.1.
USING BLOCK MODE

10.1.1.
General Overview

The purpose of the block mode interface is to allow JAM to be used with terminals, like the HP2392A and IBM 3270's, that operate in block mode. Such terminals, which are hereinafter referred to as block mode terminals, operate differently than their interactive or character mode counterparts in that they do not interact with the computer on every keystroke. Instead, a formatted screen is sent to the terminal and processed by the terminal locally. When a function key is pressed, data are transmitted to the computer and are available to the program which sent the formatted screen.

Block mode terminals typically have capabilities for defining protected and unprotected fields and sometimes allow a minimal set of character validations such as restricting a field to only allow digits. They do not provide JAM–like capabilities such as shifting, scrolling and provisions for post–field validation. It should therefore seem obvious that an application will behave slightly differently on a block mode terminal than on an interactive one. The goal of the block mode interface, however, is to minimize these differences and, to the greatest extent possible, allow applications to be created that can operate

in either mode without the need for the programmer to consider the differences. This is in keeping with the JAM philosophy of creating terminal-independent applications.

10.1.2.
Authoring

Certain JAM utilities, like `modkey`, the Screen Editor, and the Data Dictionary Editor only work in interactive mode. Thus, they can only be used with interactive terminals or those that can be switched programmatically between block and interactive mode.

`jxform` is the JAM authoring utility. It allows the user to navigate through the screens in an application and to invoke the Screen and Data Dictionary Editors when appropriate. When used with block mode-only terminals, `jxform` does not permit entry into the aforementioned utilities. When used with hybrid terminals (i.e. those that can switch between block and interactive mode programmatically), `jxform` forces interactive mode before entering the utilities.

10.1.3.
Selecting Block Mode

JAM operates with three types of terminals: interactive-only, block mode-only, and hybrid. Block mode can be used with either of the latter two.

By default, JAM operates in interactive mode regardless of the terminal type. To operate in block mode requires a block terminal driver to be linked with the system. (Block terminal drivers are described in detail later.) This alone, however, will not initiate block mode; two additional things must be done.

First there must be a call to `xsm_blkinit`. This is generally done in the "main" routine of the application, `jmain.pll`. If this call is absent, the application will be run in interactive mode. Also the additional code to support block mode will not be linked with the program. Thus programs not desiring block mode support are not penalized.

Second the correct block mode driver must be selected. This can be done in one of two ways.

If the application program author knows the correct driver he/she can install it by calling `xsm_uinstall`. This should be done before calling `xsm_blkinit`. Typically the program will install a "hard-coded" driver, but it could instead key off of SMTERM, or some other environment variable, to find the correct one. In this case the application will run in block mode, independent of the end user's preference.

The second method for selecting the driver leaves the job to the end user. If `xsm_blkinit` is called without previously installing a driver, the entry BLKDRIVER in the video

file is examined. If it is absent, `xsm_blkinit` fails and the application remains in interactive mode. If it is present the name given there is used to find the correct driver. This is done by a table lookup in a source routine (`blkdrvr.c`) that must be linked with the application. Naturally all possible choices of the driver must also be linked with the program. In this case the end user can override the application programers desire to use block mode.

The design allows for three scenarios: the programmer can prohibit block mode (no call to `xsm_blkinit`), the programmer can force block mode (`xsm_install` followed by `xsm_blkinit`), or the programmer can permit block mode but allow the end user final say (`xsm_blkinit` only).

Note that the application never calls `sm_blkdrvr`. The source code to that routine is given to customers to enable them to extend the capabilities of the second method.

10.1.4.

Differences Between Block Mode And Interactive Mode

Although every attempt has been made to preserve the look and feel of applications operating in block mode, the following differences between block mode and interactive mode should be noted.

Windows

Windows work much as they do in interactive mode. The only noticable difference is that the cursor is not be restricted to the active window as this is not possible in block mode. In keeping with the concepts of interactive mode, however, only the fields on the active window are unprotected.

Menus

In interactive mode, menus utilize a "bounce bar" to track the cursor. The bounce bar moves when cursor-positioning keys are pressed and when ascii data are typed. Since block mode terminals do not return these keys, another approach must be taken. We supply two options:

In option 1, menu fields in block mode are unprotected, making it easy for an operator to tab to them. To make a selection, the operator positions to the appropriate field and presses XMIT. Thus, selection is similar to interactive mode except there is no bounce bar and there is no provision for selecting by typing the first N characters of the menu choice.

If the operator inadvertently types over a menu field there are no adverse consequences as JAM will "remember" the contents and restore it at an appropriate time.

This approach works well since the same screens can be used for block and interactive mode operation. However, for those who do not wish to allow the operator to type over menu choice fields, option 2 may be chosen. With option 2, JAM creates an unprotected field to the left of each menu choice so the menu fields themselves can remain protected. The operator can tab to these new fields to make a selection, or type the first character of a menu field and press XMIT. The new fields to the left of the menu choices are created as long as there is room on the screen even if it means they would be placed in a border or a separate window. If there is no room on the screen because the menu field starts in position 1 or 2, the system reverts to option 1.

The above works well for traditional menus, but two–level (pull– down) menus pose a different problem in that the ONLY way to move horizontally in interactive mode is via the arrows (since TAB moves between the entries of the sub–menu). Thus, in block mode the following happens. When a pull–down menu is active, JAM unprotects all main menu fields except the one with which the pull–down is associated. Thus, the operator can either make a selection from the pull–down or tab to another main menu choice and press XMIT causing its sub–menu to be activated.

The two options for processing menus described above work equally well for pull–down menus.

Character Validation

The block mode interface takes advantage of the terminal's capabilities for character validation. However, for situations in which the specified validations go beyond what the terminal can handle, JAM will validate the character data during Screen Validation. The result will be something like this:

The operator enters alphabetic data in a digits–only field. When the XMIT key is pressed, all fields are validated in the normal fashion, left–to–right, top–to–bottom. Thus, the cursor will be positioned to the errant field and a message displayed.

Since programs do not rely on data being correct unless and until Screen Validation completes without error, this should pose no problem. The only consideration is that invalid character data can get into the screen buffer and LDB if the operator enters incorrect characters and then presses something like EXIT (this cannot happen in interactive mode because the invalid characters would not be allowed in the first place.

The only reason for mentioning this has to do with how punctuation characters in digits–only fields are handled. Let's say that a digits–only field got filled with slash ("/") characters and this, in turn, got transferred to the screen buffer and hence to the LDB. On a subsequent attempt to enter data into the field, an attempt to merge the slashes with the

entered data would be made. But since the field has ALL slash characters, there would be no room for the digits.

Thus, to eliminate the possibility of "punctuation character creep", when reading data from a digits-only field, JAM first strips out all punctuation characters from the field and then merges in the punctuation characters from the screen buffer.

Field Validation

Clearly, fields are not validated when TAB and RETURN are pressed as in interactive mode. Thus, like character validations, field validations will be deferred until Screen Validation. This should not be a problem since, even in interactive mode, the operator can usually bypass field validation by using the arrow keys to move from field to field. Therefore, programs should not rely on the data until Screen Validation passes without error in either mode.

One type of field validation is worth noting. Consider a field with an attached function which does a database lookup and displays information in another field. In interactive mode, this would usually be executed when the field is completed, so the user would see the result. Since this is not really a validation, deferring it until Screen Validation would not help because the data would never be seen by the operator. Therefore, if this type of feature is contemplated in a block mode environment, the database lookup should be attached to a function key rather than as an attached function.

Screen Validation

Screen validation works the same in interactive and block mode. The cursor will be positioned to the first field in error and a message will be displayed to the operator.

Right Justified Fields

Unless the block mode terminal supports this feature directly, the cursor will always be positioned to the left side of right justified fields when the cursor enters them.

Field Entry Function, Automatic Help, Status Text, etc.

These are disabled in block mode since JAM does not know when fields are entered.

Currency Fields

Currency edits are usually applied to fields as they are exited. In block mode, since this is not possible, currency formatting is done during screen validation. Care should be taken

with right justified currency formats since subsequent entry may be difficult for the reasons cited above in the section on right justified fields.

Shifting Fields

Normally fields shift when the left or right arrows are pressed with the cursor at the start or end of a shifting field or, in the case of unprotected fields, when the operator types off the edge of the field. Since arrows and data entry keys are not returned in block mode, this is not possible. To utilize shifting fields in block mode, use the logical keys: Shift Left and Shift Right. These shift the field by the shifting increment and work equally well in block and interactive mode.

An alternative is to use the Zoom feature if all shifting fields are limited to the width of the screen.

Scrolling Fields

This is similar to the situation with shifting fields. In block mode, one can define function keys as PAGE UP and PAGE DOWN, or use the Zoom feature.

Messages

Error messages are normally acknowledged by pressing the space bar, although the specific key used can vary depending on the setting of error message options. Also, options govern whether the key should be used as the next keystroke or discarded after the message is acknowledged. In block mode, ANY key that gets transmitted from the terminal will suffice to acknowledge messages, regardless of what key is defined for that purpose. Using or discarding the acknowledgement key apply equally to block mode and interactive mode.

With query messages, JAM normally expects a Y or N response. In block mode, JAM will create a field on the status line into which the Y or N response can be entered. This entry must be followed by the XMIT key for it to be accepted. On terminals that have a separate stauts line it is not possible to create such a field. In these cases, XMIT will be treated as a positive response; EXIT will be treated as a negative response.

Insert Mode

Insert mode will operate in whatever way the block mode terminal supports. However, since JAM never knows if insert mode is set or not in block mode, it will, for terminals in which this is a problem, reset insert mode before transmitting data to the terminal. This is so the new data will not be INSERTED into the terminal buffer, causing all other data to move around.

Non–Display Fields

If the block mode terminal supports this feature, it will be used.

System Calls

These operate as in interactive mode. However, before passing control to the OS, JAM sets the terminal to the mode (block or interactive) expected by the OS, and resets it upon return from the system call. The JAM routines `xsm_leave` and `xsm_return` do the same.

Zoom

With the exception of the limitations expressed in the sections on shifting and scrolling, Zoom works as in interactive mode.

Help and Item Selection

With the exception of the limitations expressed in the sections on shifting, scrolling, field entry and menu processing, these functions work as in interactive mode.

Groups

Radio buttons and check lists behave similar to menus as described above.

10.2. WRITING A BLOCK MODE DRIVER

10.2.1. Installation

There are two parts to the installation process. These were discussed in greater detail above.

First a block terminal driver must be installed. This driver performs the low level communication between JAM and the terminal. The PL/1 interface does not currently support writing your own block mode drivers.

Next the application program must initiate block mode by making the appropriate subroutine call. The application program can also switch to interactive mode by means of a call. The assumption is that the default is interactive mode, thus a call to set block mode is needed even if that is the normal mode of the operating system. The application program can also set some operating parameters by means of a subroutine call.

10.2.2.

Application Program Support

JAM programs assume that the terminal is in interactive mode. Explicit calls are needed to switch from interactive to block and vice versa. To turn on block mode, the program should call `xsm_blkinit`. To turn off block mode (and turn on interactive mode) the program calls `xsm_blkreset`. The Screen Editor The key mapping utility (`modkey`) also requires interactive mode. The authoring utility (`jxform`) can be made to work in block mode, switching to interactive mode when the Screen Editor is invoked. This can be done by inserting the appropriate calls in `jxmain.pl1` (provided) and relinking `jxform`.

The routine `xsm_option` can be used to set some user-preference items.

Chapter 11.
Library Function Overview

In this chapter, we summarize the JAM library functions and list them in categories. All JAM library function names begin with the prefix `xsm_`. However, in the Function Reference Chapter and in this chapter, the functions are listed without prefix for clarity.

In addition to stripping off the prefix in the listings that follow, groups of closely related variant functions are listed under a single root name. The functions `xsm_r_form`, `xsm_d_form`, and `xsm_l_form`, for example, are all grouped under the heading `form`. In a few cases, functions may be listed under a name that is not a portion of the the function name but is suggestive of the utility of the function. For example, the function `xsm_r_at_cur`, which displays a window at the cursor position, is listed under the root name `window`, along with `xsm_r_window` (which displays a window at a fixed location) and a number of other window display routines. The calling syntax of each function is found in the SYNOPSIS section of the function listing in the Function Reference Chapter.

Most JAM library routines fall into one of the following categories:

- Initialization/Reset
- Screen and Viewport Control
- Keyboard and Display I/O
- Field/Array Data Access
- Field/Array Characteristic Access
- Group Access
- Local Data Block Access
- Cursor Control
- Message Display

- Scrolling and Shifting
- Mass Storage and Retrieval
- Validation
- Global Data and Changing JAM's Behavior
- Soft Keys and Keysets
- JAM Executive Control
- Block Mode Control
- Miscellaneous

The following sections summarize the functions that fall into these categories. Some listings are found in more than one category.

11.1. INITIALIZATION/RESET

The following library functions are called in order to initialize or reset certain aspects of the JAM runtime environment. Those that are necessary for the proper operation of JAM are called from within the supplied `main` routine source modules `jmain.pl1` and `jxmain.pl1`.

`cancel`	reset the display and exit
`dicname`	set data dictionary name
`ininames`	record names of initial data files for local data block
`initcrt`	initialize the display and JAM data structures
`keyinit`	initialize key translation table
`ldb_init`	initialize (or reinitialize) the local data block
`leave`	prepare to leave a JAM application temporarily
`msgread`	read message file into memory
`resetcrt`	reset the terminal to operating system default state
`return`	prepare for return to JAM application
`vinit`	initialize video translation tables

11.2. SCREEN AND VIEWPORT CONTROL

The following routines are used to control viewports, the display of screens, and the form and window stacks.

close_window	close current window
form	display a screen as a form
hlp_by_name	display help window
issv	determine if a screen in the saved list
jclose	close current window or form under JAM Executive control
jform	display a screen as a form under JAM control
jwindow	display a window at a given position under JAM control
mwindow	display a status message in a window
shrink_to_fit	remove trailing empty array elements and shrink screen
sibling	define the current window as being or not being a sibling window
submenu_close	close the current submenu
svscreen	register a list of screens on the save list
unsvscreen	remove screens from the save list
viewport	modify viewport size and offset
wcount	obtain number of currently open windows
wdeselect	restore the formerly active window
window	display a window at a given position
winsize	allow end-user to interactively move and resize a window
wselect	activate a window

11.3.
DISPLAY TERMINAL I/O

The following routines provide the interface to JAM terminal I/O.

bel	beep!
bkrect	set background color of rectangle
do_region	rewrite part or all of a screen line
flush	flush delayed writes to the display
getkey	get logical value of the key hit
input	open the keyboard for data entry and menu selection
keyfilter	control keystroke record/playback filtering
keyhit	test whether a key has been typed ahead

`keylabel`	get the printable name of a logical key
`keyoption`	set cursor control key options
`m_flush`	flush the message line
`rescreen`	refresh the data displayed on the screen
`resize`	dynamically change the size of the display
`ungetkey`	push back a translated key on the input

11.4.
FIELD/ARRAY DATA ACCESS

The following routines access the data in fields and arrays. Most routines in this section have a number of variants that perform the same task but reference the field to be accessed differently. In these cases, the calling syntax of the *major* variant is listed under the SYNOPSIS section of the listing in the Function Reference Chapter. All other variants are listed under the VARIANTS section.

Most field access routines have five variants, although some have fewer. The five possible variants are shown in the table below:

Variants of Functions That Access Fields		
Prefix	*Example*	*Description*
`xsm_`	`xsm_intval(fieldnum);`	Access a field via field number.
`xsm_n_`	`xsm_n_intval(fieldname);`	Access a field (or an entire array) via field name. Access the LDB if there is no field on the screen.
`xsm_i_`	`xsm_i_intval(fieldname, occurrence);`	Access an occurrence via field name and occurrence number. Access the LDB if there is no field on the screen.
`xsm_o_`	`xsm_o_intval(fieldnum, occurrence);`	Access an occurrence via field number and occurrence number.
`xsm_e_`	`xsm_e_intval(fieldname, element);`	Access an element via field name and element number.

amt_format	write data to a field, applying currency editing
calc	execute a math edit style expression
cl_unprot	clear all unprotected fields
clear_array	clear all data in an array
dblval	get the value of a field as a real number
dlength	get the length of a field's contents
doccur	delete occurrences
dtofield	write a real number to a field
fptr	get the content of a field
getfield	copy the contents of a field
gwrap	get the contents of a wordwrap array
intval	get the integer value of a field
ioccur	insert blank occurrences into an array
is_no	test field for no
is_yes	test field for yes
itofield	write an integer value to a field
lngval	get the long integer value of a field
ltofield	place a long integer in a field
null	test if field is null
putfield	put a string into a field
pwrap	put text to a wordwrap field
strip_amt_ptr	strip amount editing characters from a string

11.5. FIELD/ARRAY ATTRIBUTE ACCESS

The following routines access information about fields and arrays. Like the routines in the previous section on field and array data access, each of these routines generally have five distinct variants. See the discussion in the introduction to the previous section for more information on variants of JAM library functions that access fields.

base_fldno	get the field number of the first element of an array
bitop	manipulate validation and data editing bits
chg_attr	change the display attribute of a field

`cl_all_mdts`	clear all MDT bits
`dlength`	get the length of a field's contents
`edit_ptr`	get special edit string
`finquire`	obtain information about a field
`fldno`	get the field number of an array element or occurrence
`ftog`	convert field references to group references
`ftype`	get the data type and precision of a field
`gtof`	convert a group name and index into a field number and occurrence
`length`	get the maximum length of a field
`max_occur`	get the maximum number of occurrences
`name`	obtain field name given field number
`num_occurs`	find the highest numbered occurrence containing data
`protect`	protect an array
`sc_max`	alter the maximum number of items allowed in a scrollable array
`size_of_array`	get the number of elements
`tst_all_mdts`	find first modified occurrence in the screen

11.6. GROUP ACCESS

The following routines access groups, that is, radio buttons and check lists. Groups are made up of fields that have attributes and data in them, but groups in and of themselves are implemented as phantom fields which take up no screen real estate. The value of a group indicates the set of selected consituent fields, although it is not recommended that that value ever be accessed or modified directly with any of the field access routines discussed in the preceding sections.

The routines that follow are those that are recommended for accessing groups:

`deselect`	deselect a checklist occurrence
`ftog`	convert field references to group references
`gp_inquire`	obtain information about a group
`gtof`	convert a group name and index into a field number and occurrence

`isselected`	determine whether a radio button or checklist occurrence has been selected
`select`	select a checklist or radio button occurrence

11.7. LOCAL DATA BLOCK ACCESS

The following routines access the Local Data Block, or LDB. Note that any of the field data access routines that reference fields by name or name and occurrence number (eg `xsm_n` and `xsm_i_` variants) will access the LDB if the named field does not exist on the active screen.

`allget`	load screen from the LDB
`dicname`	set data dictionary name
`dd_able`	turn LDB write–through on or off
`ininames`	record names of initial data files for local data block
`lclear`	erase LDB entries of one scope
`ldb_init`	initialize (or reinitialize) the local data block
`lreset`	reinitialize LDB entries of one scope
`lstore`	copy everything from screen to LDB

11.8. CURSOR CONTROL

The following routines control the positioning and display of the cursor on the active screen.

`ascroll`	scroll to a given occurrence
`backtab`	backtab to the start of the last unprotected field
`c_off`	turn the cursor off
`c_on`	turn the cursor on
`c_vis`	turn cursor position display on or off
`disp_off`	get displacement of cursor from start of field
`getcurno`	get current field number
`gofield`	move the cursor into a field

`home`	home the cursor
`last`	position the cursor in the last field
`nl`	position cursor to the first unprotected field beyond the current line
`occur_no`	get the current occurrence number
`off_gofield`	move the cursor into a field, offset from the left
`rscroll`	scroll an array
`sh_off`	determine the cursor location relative to the start of a shifting field
`tab`	move the cursor to the next unprotected field

11.9. MESSAGE DISPLAY

The following routines are intended for the access and display of runtime application messages.

`d_msg_line`	display a message on the status line
`emsg`	display an error message and reset the message line, without turning on the cursor
`err_reset`	display an error message and reset the status line
`m_flush`	flush the message line
`msg`	display a message at a given column on the status line
`msg_get`	find a message given its number
`msgfind`	find a message given its number
`msgread`	read message file into memory
`mwindow`	display a status message in a window
`query_msg`	display a question, and return a yes or no answer
`qui_msg`	display a message preceded by a constant tag, and reset the message line
`quiet_err`	display error message preceded by a constant tag, and reset the status line
`setbkstat`	set background text for status line
`setstatus`	turn alternating background status message on or off

11.10.
SCROLLING AND SHIFTING

The following routines provide access to shifting and scrolling fields and arrays.

achg	change the display attribute of an occurrence within a scrolling array
ascroll	scroll to a given occurrence
doccur	delete occurrences
ioccur	insert blank occurrences into an array
max_occur	get the maximum number of occurrences
num_occurs	find the highest numbered occurrence containing data
oshift	shift a field by a given amount
rscroll	scroll an array
sc_max	alter the maximum number of items allowed in a scrollable array
sh_off	determine the cursor location relative to the start of a shifting field
t_scroll	test whether an array can scroll
t_shift	test whether field can shift
tst_all_mdts	find first modified occurrence

11.11.
MASS STORAGE AND RETRIEVAL

The following routines move data to or from sets of fields in the screen or LDB.

rd_part	read part of a data structure to the current screen
rdstruct	read data from a structure to the screen
restore_data	restore previously saved data to the screen
rrecord	read data from a structure to a data dictionary record
wrecord	write data from a data dictionary record to a structure
wrt_part	write part of the screen to a structure
wrtstruct	write data from the screen to a structure

11.12. VALIDATION

The following routines provide an application interface to the field and group validation processes.

`bitop`	manipulate validation and data editing bits
`ckdigit`	validate check digit
`fval`	force field validation
`gval`	force group validation
`novalbit`	forcibly invalidate a field
`s_val`	validate the current screen

11.13. GLOBAL DATA AND CHANGING JAM'S BEHAVIOR

The following routines grant access to global data and provide a way to manipulate certain aspects of JAM and Screen Manager behavior.

`async`	install an asynchronous function
`dd_able`	turn LDB write-through on or off
`finquire`	obtain information about a field
`gp_inquire`	obtain information about a group
`inquire`	obtain value of a global integer variable
`isabort`	test and set the abort control flag
`iset`	change value of integer global variable
`keyfilter`	control keystroke record/playback filtering
`keyoption`	set cursor control key options
`li_func`	install an application hook function
`msgread`	read message file into memory
`option`	set a Screen Manager option
`pinquire`	obtain value of a global strings
`pset`	Modify value of global strings

`resize`	dynamically change the size of the display
`uinstall`	install an application function

11.14. SOFT KEYS AND KEYSETS

The following routines provide an application interface to JAM's soft key support.

`c_keyset`	close a keyset
`keyset`	open a keyset
`kscscope`	query current keyset scope
`ksinq`	inquire about key set information
`ksoff`	turn off key labels
`kson`	turn on key labels
`skinq`	obtain soft key information by position
`skmark`	mark or unmark a softkey label by position
`skset`	set characteristics of a soft key by position
`skvinq`	obtain soft key information by value
`skvmark`	mark a soft key by value
`skvset`	set characteristics of a soft key by value

11.15. JAM EXECUTIVE CONTROL

The following routines, available only to applications using the JAM Executive, provide JAM Executive services.

`getjctrl`	get control string associated with a key
`jclose`	close current window or form under JAM Executive control
`jform`	display a screen as a form under JAM control
`jtop`	start the JAM Executive
`jwindow`	display a window at a given position under JAM control
`putjctrl`	associate a control string with a key

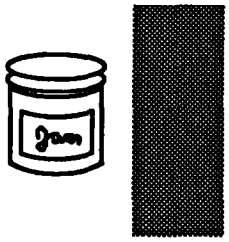
11.16.
BLOCK MODE CONTROL

The following routines are used in applications requiring block mode support.

`blkdrvr`	install block mode driver
`blkinit`	initialize (and turn on) block mode terminal
`blkreset`	reset (and turn off) block mode terminal

11.17.
MISCELLANEOUS

`fi_path`	return the full path name of a file
`formlist`	update list of memory–resident files
`jplcall`	execute a JPL procedure
`jplload`	execute the JPL load command
`jplpublic`	execute the JPL public command
`jplunload`	execute the JPL unload command
`l_close`	close a library
`l_open`	open a library
`rmformlist`	empty the memory–resident form list
`sdtime`	get formatted system date and time
`udtime`	format user–supplied date and time



Chapter 12.
Function Reference

All JAM function names begin with the prefix `xsm_`. In the Function Reference Chapter functions are listed without the prefix and, in a few cases, under a name that is not a portion of the function name — but that is suggestive of the utility of the function. For example, the function `xsm_r_at_cur`, which displays a window at a specified position, is found under the listing name `window`, along with the function `xsm_r_window`. In these cases, the calling syntax of each function is listed under the SYNOPSIS section of the listing.

For each entry, you will find several sections:

- A synopsis similar to a PL/1 function declaration, giving the types of the arguments and return value.
- A description of the function's arguments, prerequisites, results, and side-effects.
- The function's return values, if any, and their meanings.
- A list of variants.
- A list of functions that perform related tasks.
- An example illustrating the function's use.

A routine that calls JAM functions should include the file `smdefs.incl.pl1`. If another file should be included, then it is referenced in the synopsis section.

To view functions by category, refer to the Library Function Overview (chapter 11.) To view a complete list of functions alphabetically by the actual function name (including the `xsm_` prefix), see the Library Function Index (chapter 13.).

achg

change the display attribute of an occurrence within a scrolling array

SYNOPSIS

```
declare field_number       fixed binary(31);
declare occurrence         fixed binary(31);
declare display_attribute  fixed binary(31);
declare status             fixed binary(31);
status = xsm_o_achg(field_number, occurrence,
         display_attribute);
```

DESCRIPTION

NOTE: This function has only two variants, `xsm_o_achg` and `xsm_i_achg`. There is NO `xsm_achg`.

This function changes the display attribute of an occurrence within a scrollable array. If the occurrence is onscreen, the attribute with which the occurrence is currently displayed is changed as well. When the occurrence is scrolled to another position within the array the new attribute moves with the occurrence. Use `xsm_chg_attr` if you want all of the occurrences within the array to scroll through an attribute so that their appearance is determined by their onscreen positions.

Possible values for the argument `display_attribute` are defined in the header file `smdefs.incl.pl1`, as shown in the table below:

Foreground Attributes	*Background Attributes*
BLANK	B_HILIGHT
REVERSE	
UNDERLN	
BLINK	
HILIGHT	
STANDOUT	
DIM	
ACS (alternate character set)	

Foreground Colors	*Background Colors*
BLACK	B_BLACK
BLUE	B_BLUE
GREEN	B_GREEN
CYAN	B_CYAN
RED	B_RED
MAGENTA	B_MAGENTA
YELLOW	B_YELLOW
WHITE	B_WHITE

Foreground colors may be used alone or with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

If `display_attribute` is zero, the occurrence's display attribute is removed, leaving it with the field display attribute. Then, if that occurrence is onscreen, it is displayed with the attribute attached to its field.

This function will not work on an array that is not scrollable. Use `xsm_chg_attr` to change the display attribute of an individual field.

RETURNS

–1 if the field isn't found or isn't scrollable, or if `occurrence` is invalid. 0 otherwise.

VARIANTS

```
status = xsm_i_achg(field_name,. occurrence, display_attribute);
```

RELATED FUNCTIONS

```
status = xsm_chg_attr(field_number, display_attribute);
```

allget
load screen from the LDB

SYNOPSIS

```
declare respect_flag       fixed binary(31);
call xsm_allget(respect_flag);
```

DESCRIPTION

This function copies data from the local data block to fields on the current screen with matching names.

If `respect_flag` is nonzero, this function does not write to fields that already contain data, or that have their MDT bits set. If the flag is zero, all fields are initialized. When this function is called by the JAM run-time system, or by your screen entry function, it does *not* set MDT bits for the fields it initializes.

This function is called automatically by the JAM screen-display logic, unless LDB processing has been turned off using `xsm_dd_able`. Application code should not normally need to call it.

RELATED FUNCTIONS

```
call xsm_dd_able(flag);
status = xsm_lstore();
```

amt_format

write data to a field, applying currency editing

SYNOPSIS

```
declare field_number        fixed binary(31);
declare buffer              char(256) varying;
declare status              fixed binary(31);
status = xsm_amt_format(field_number, buffer);
```

DESCRIPTION

If the specified field has a currency edit, it is applied to the data in `buffer`. If the resulting string is too long for the field, an error message is displayed. Otherwise, `xsm_putfield` is called to write the edited string to the specified field.

If the field has no currency edit, `xsm_putfield` is called with the unedited string.

RETURNS

–1 if the field is not found or the occurrence is out of range;
–2 if the edited string will not fit in the field;
0 otherwise.

VARIANTS

```
status =  xsm_e_amt_format(field_name, element, buffer);
status = xsm_i_amt_format(field_name, occurrence, buffer);
status = xsm_n_amt_format(field_name, buffer);
status = xsm_o_amt_format(field_number, occurrence, buffer);
```

RELATED FUNCTIONS

```
status = xsm_dtofield(field_number, value, format);
outbuf = xsm_strip_amt_ptr(field_number, inbuf, );
```

ascroll

scroll to a given occurrence

SYNOPSIS

```
declare field_number        fixed binary(31);
declare occurrence          fixed binary(31);
declare status              fixed binary(31);
status = xsm_ascroll(field_number, occurrence);
```

DESCRIPTION

This function scrolls the designated field so that the indicated `occurrence` appears there. Synchronized arrays will scroll along with the target array.

The field need not be the first element of a scrolling array. You can use this function, for instance, to place the nineteenth occurrence in the third onscreen element of a five-element scrolling array.

The validity of certain combinations of parameters depends on the exact nature of the field. For instance, if field number 7 is the third element of a scrolling array and `occurrence` is 1 a call to `xsm_ascroll` will fail on a non-circular scrolling array but succeed if scrolling is circular.

RETURNS

-1 if field or occurrence specification is invalid,
0 otherwise.

VARIANTS

```
status = xsm_n_ascroll(field_name, occurrence);
```

RELATED FUNCTIONS

```
lines = xsm_rscroll(field_number, req_scroll);
status = xsm_t_scroll(field_number);
```

async

install an asynchronous function

SYNOPSIS

```
declare func              entry variable;
declare timeout           fixed binary(31);
call xsm_async(func, timeout);
```

DESCRIPTION

This routine installs a a function that will be called regularly during keyboard processing (ie. - `xsm_input`). The first parameter is the address of the function. Use the operating system subroutine `s$find_entry` to find the entry point. The second parameter is the timeout, in tenths of a second, between subsequent function calls.

The asynchronous function is called only when the keyboard is being read, and only if a keystroke does not arrive within the specified timeout. The authoring utility, `jxform`, uses an asynchronous function to update its cursor position display. An asynchronous function might also be used to implement a real-time clock display.

RELATED FUNCTIONS

```
status = xsm_uinstall( usage, func, func_name);
```

backtab

backtab to the start of the last unprotected field

SYNOPSIS

```
call xsm_backtab();
```

DESCRIPTION

When the cursor is in a field unprotected from tabbing into, but not in the first enterable position, it is moved to the first enterable position of that field. However, if the cursor is in a field with a previous–field edit and one of the fields specified by the edit is unprotected from tabbing, the cursor is moved to the first enterable position of that field. Otherwise, the cursor is moved to the first enterable position of the tab–unprotected field with the next lowest field number. If the cursor is in the first position of the first unprotected field on the screen, or before the first unprotected field on the screen, it wraps backward into the last unprotected field. When there are no unprotected fields, the cursor doesn't move.

If the destination field is shiftable, it is reset according to its justification. The first enterable position depends on the justification of the field and, in fields with embedded punctuation, on the presence of punctuation.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

This function is called when the JAM logical key BACK is struck.

RELATED FUNCTIONS

```
field_number = xsm_home();
call xsm_last();
call xsm_nl();
call xsm_tab();
```

base_fldno

get the field number of the first element of an array

SYNOPSIS

```
declare field_number       fixed binary(31);
declare base_number        fixed binary(31);
base_number = xsm_base_fldno(field_number);
```

DESCRIPTION

A base field number is the field number of the first element of an array. Use `xsm_base_fldno` to obtain the base field number of an array.

RETURNS

The field number of the base element of the array containing the specified field, or 0 if the field number is out of range.

bel

beep!

SYNOPSIS

```
call xsm_bel();
```

DESCRIPTION

Causes the terminal to beep, ordinarily by transmitting the ASCII BEL code to it. If there is a BELL entry in the video file, xsm_bel will transmit that instead, usually causing the terminal to flash instead of beeping.

Even if there is no BELL entry, use this function instead of sending a BEL, because certain displays use BEL as a graphics character.

Including a **%B** at the beginning of a message displayed on the status line will cause this function to be called.

bitop

manipulate validation and data editing bits

SYNOPSIS

```
%include 'smbitops.incl.pl1';

declare field_number        fixed binary(31);
declare action              fixed binary(31);
declare bit                 fixed binary(31);
declare status              fixed binary(31);
status = xsm_bitop(field_number, action, bit);
```

DESCRIPTION

You can use this function to inspect and modify validation and data editing bits of screen fields, without reference to internal data structures. The first parameter identifies the field to be operated upon.

`action` can include a test and at most one manipulation from the following table of mnemonics, which are defined in `smbitops.incl.pl1`:

Mnemonic	*Meaning*
BIT_CLR	Turn bit off
BIT_SET	Turn bit on
BIT_TOGL	Flip state of bit
BIT_TST	Report state of bit

The third parameter is a bit identifier, drawn from the following table:

Character edits				
N_ALL	N_DIGIT	N_YES_NO	N_ALPHA	N_NUMERIC
N_ALPHNUM	N_FCMASK			

Field edits	*Field edits*			
N_RTJUST	N_REQD	N_VALIDED	N_MDT	N_CLRINP
N_MENU	N_UPPER	N_LOWER	N_RETENTRY	N_FILLED
N_NOTAB	N_WRAP	N_ADDLEDS	N_EPROTECT	N_TPROTECT
N_CPROTECT	N_VPROTECT	N_ALLPROTECT	N_SELECTED	

The character edits are not, strictly speaking, bits; you cannot toggle them, but the other functions work as you would expect. N_ALLPROTECT is a special value meaning all four protect bits at once.

N_VALIDED and N_MDT are the only bit operations that can apply to individual offscreen and onscreen occurrences. The protection operations can apply to an array as a whole, including offscreen occurrences (see xsm_aprotect). All other bit operations are attached to fixed onscreen positions.

The variants xsm_e_bitop and xsm_n_bitop can take a group name as an argument. The function will then affect the group bits.

This function has two additional variants, xsm_a_bitop and xsm_t_bitop, which perform the requested bit operation on all elements of an array. Their synopsis appear below. If you include BIT_TST, these variants return 1 only if bit is set for *every* element of the array. The variants xsm_i_bitop and xsm_o_bitop are restricted to N_VALIDED and N_MDT.

RETURNS

1 if there was no error, the action included
-1 if the field or occurrence cannot be found
-2 if the action or bit identifiers are invalid; a test operation, and bit was set
-3 if xsm_i_bitop or xsm_o_bitop was called with bit set to something other than N_VALIDED or N_MDT
0 otherwise.

VARIANTS

```
status = xsm_a_bitop(array_name, action, bit);
status = xsm_e_bitop(array_name, element, action, bit);
status = xsm_i_bitop(array_name, occurrence, action, bit);
status = xsm_n_bitop(name, action, bit);
status = xsm_o_bitop(field_number, occurrence, action, bit);
status = xsm_t_bitop(array_number, action, bit);
```

bkrect

set background color of rectangle

SYNOPSIS

```
  declare start_line        fixed binary(31);
  declare start_column      fixed binary(31);
  declare num_of_lines      fixed binary(31);
  declare number_of_columns fixed binary(31);
  declare background_colors fixed binary(31);
  declare status            fixed binary(31);
" status = xsm_bkrect(start_line, start_column, num_of_lines,
          number_of_columns, background_colors);
```

DESCRIPTION

This function changes the background color of a rectangular area of the current screen. Any fields or elements that begin within the rectangular area will have their background attributes changed to the specified attribute. This means that if there are any fields or elements that are not entirely contained within the rectangular area, a ragged edge will result. Display text that falls with in the rectangular area will have its background attribute set.

The arguments `start_line` and `start_column` can have any value from 1 through the number of lines (or columns) on the screen.

The background color must be one of the mnemonics defined in `smdefs.incl.pl1` (B_BLACK, B_BLUE, etc.). You can highlight the background color by oring the background color attribute with B_HILIGHT.

RETURNS

-1 if the starting line or column was invalid.
1 if the starting line and column were valid, but the rectangle had to be truncated to fit.
0 if no error.

blkinit

initialize (and turn on) block mode terminal

SYNOPSIS

```
declare return_value       fixed binary(31);
return_value = xsm_blkinit();
```

DESCRIPTION

This routine must be called by the application program to initiate block mode terminal action. A block mode terminal driver must have been previously installed.

This routine checks that a block mode terminal driver is installed. If a driver is found, it is called. The driver should return 0 if all is successful.

Generally the return code can be ignored. If the terminal cannot be put into block mode it will still work (possibly better) in interactive mode.

If the driver signifies that all is OK, the global variable `sm_blkcontrol` is set to point to the local block terminal control handler. All Screen Manager calls for block mode support are made through this control routine.

On the first call to the present routine the driver is called with BLK_INIT to perform any required initialization.

On subsequent calls BLK_BLOCK is called instead of BLK_INIT.

RETURNS

return value from driver if one exists.
–1 otherwise.

RELATED FUNCTIONS

```
return_value = xsm_blkreset();
```

blkreset

reset (and turn off) block mode terminal

SYNOPSIS

```
declare return_value       fixed binary(31);
return_value = xsm_blkreset();
```

DESCRIPTION

This routine must be called by the application program to reset block mode terminal action. A block mode terminal driver must have been previously installed.

This routine checks that a block mode terminal driver is installed. If a driver is found, it is called. The driver should return 0 if all is successful.

Generally the return code can be ignored as the terminal is often already in interactive mode. The exception is on those systems that are normally block mode. Many JAM programs rely on the fact that the terminal can be put into interactive mode.

Note that the driver is called with BLK_CHAR, not with BLK_RESET. The only time the driver is called for a full reset is when JAM is about to go to the operating system – either exiting or performing a "shell escape".

RETURNS

return value from driver if one exists.
–1 otherwise.

RELATED FUNCTIONS

```
return_value = xsm_blkinit();
```

c_keyset
close a keyset

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

declare scope                fixed binary(31);
declare status               fixed binary(31);
status = xsm_c_keyset(scope);
```

DESCRIPTION

This function closes the keyset of the given scope. It frees all memory associated with the keyset and marks that scope as free. If the keyset was currently displayed, the keyset labels are changed to reflect the new keyset.

See the keyset chapter of the Author's Guide for a detailed explanation of keyset scopes.

Scope Mnemonic from `smsoftk.incl.pl1`	*Description*
KS_APPLIC	Application scope.
KS_FORM	Form or window scope.
KS_SYSTEM	`jxform` system key sets.

Use `xsm_d_keyset` and `xsm_r_keyset` to open keysets.

RETURNS

0 if there is no error
–2 if there is no keyset currently at that scope
–3 if the scope is out of range

RELATED FUNCTIONS

```
status = xsm_r_keyset(name, scope);
status = xsm_d_keyset(address, scope);
```

c_off

turn the cursor off

SYNOPSIS

```
call xsm_c_off();
```

DESCRIPTION

This function notifies JAM that the normal cursor setting is off. The normal setting is in effect except:

- When a block cursor is in use, as during menu processing, the cursor is off.
- While Screen Manager functions are writing to the display the cursor is off.
- Within certain error message display functions the cursor is on.

If the display cannot turn its cursor on and off (V_CON and V_COF entries are not defined in the video file), this function will have no effect.

Use `xsm_c_on` to turn the cursor on.

RELATED FUNCTIONS

```
call xsm_c_on();
```

c_on

turn the cursor on

SYNOPSIS

```
call xsm_c_on();
```

DESCRIPTION

This function notifies JAM that the normal cursor setting is on. The normal setting is in effect except:

- When a block cursor is in use, as during menu processing, the cursor is off.
- While Screen Manager functions are writing to the display the cursor is off.
- Within certain error message display functions the cursor is on.

If the display cannot turn its cursor on and off (V_CON and V_COF entries are not defined in the video file), this function will have no effect.

Use `xsm_c_off` to turn the cursor off.

RELATED FUNCTIONS

```
call xsm_c_off();
```

c_vis

turn cursor position display on or off

SYNOPSIS

```
declare display          fixed binary(31);
call xsm_c_vis(display);
```

DESCRIPTION

Assigning a non–zero value to `display` displays subsequent status line messages with the cursor's position display. This includes background status messages. Messages that would overlap the cursor position display are truncated.

Setting `display` to zero will cause subsequent status line messages to be displayed without the cursor's position display.

This function will have no effect if the `CURPOS` entry in the video file is not defined. In that case the cursor position display will never appear.

JAM uses an asynchronous function and a status line function to perform the cursor position display. If the application has previously installed either of those, this function will override it.

calc

execute a math edit style expression

SYNOPSIS

```
declare field_number        fixed binary(31);
declare occurrence          fixed binary(31);
declare expression          char(256) varying;
declare status              fixed binary(31);
status = xsm_calc(field_number, occurrence, expression);
```

DESCRIPTION

Use `xsm_calc` to execute a math edit style expression. With this function you can perform mathematical operations that use the contents of one or more fields and then insert the result into a field.

The third parameter `expression` is a math edit style expression. See the JAM Author's Guide for a complete description on how to create the expression.

The first two parameters, `field_number` and `occurrence` identify the field and occurrence with which the calculation is associated. Normally you will not need to use them and should set them both to 0.

If you want to use relative references to fields in your expression, use the arguments `field_number` and `occurrence` to specify the field to which they should be relative.

If in the event of a math error you want the cursor to move a specific field, specify that field with `field_number`. In addition, if the desired field is an occurrence within an array, specifying the `occurrence` will cause the referenced array to scroll to `field_number`.

RETURNS

-1 is returned if a math error occurred.
0 is returned otherwise.

cancel

reset the display and exit

SYNOPSIS

```
declare arg                fixed binary(31);
call xsm_cancel(arg);
```

DESCRIPTION

This function is installed by `xsm_initcrt` to be executed if a keyboard interrupt occurs. It calls `xsm_resetcrt` to restore the display to the operating system's default state, and exits to the operating system.

If your operating system supports it, you can also install this function to handle conditions that normally cause a program to abort. If a program aborts without calling `xsm_resetcrt`, you may find your terminal in an odd state; `xsm_cancel` can prevent that.

The argument `arg` is a dummy argument. It should have the value zero.

chg_attr

change the display attribute of a field

SYNOPSIS

```
declare field_number      fixed binary(31);
declare display_attribute fixed binary(31);
declare status            fixed binary(31);
status = xsm_chg_attr(field_number, display_attribute);
```

DESCRIPTION

Use this function to change the display attribute of an individual field or an element within an array. To change an occurrence attribute so that the attribute moves with the occurrence use `xsm_o_achg`.

If the field is part of a scrolling array, then each occurrence may also have a display attribute that overrides the field display attribute when the occurrence arrives onto the screen.

Possible values for `display_attribute` are defined in `smdefs.incl.pl1`, as shown in the table below:

Foreground Attributes	*Background Attributes*
BLANK	B_HILIGHT
REVERSE	
UNDERLN	
BLINK	
HILIGHT	
STANDOUT	
DIM	
ACS (alternate character set)	
Foreground Colors	*Background Colors*
BLACK	B_BLACK
BLUE	B_BLUE
GREEN	B_GREEN

Foreground Colors	*Background Colors*
CYAN	B_CYAN
RED	B_RED
MAGENTA	B_MAGENTA
YELLOW	B_YELLOW
WHITE	B_WHITE

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

NOTE: The variant `xsm_o_chg_attr` does not take the usual arguments. The second argument is an element rather than an occurrence.

RETURNS

-1 if the field is not found
0 otherwise.

VARIANTS

```
status = xsm_e_chg_attr(field_name, element,
         display_attribute);
status = xsm_n_chg_attr(field_name, display_attribute);
status = xsm_o_chg_attr(field_number, element,
         display_attribute);
```

RELATED FUNCTIONS

```
status = xsm_o_achg(field_number, occurrence,
         display_attribute);
```

ckdigit

validate check digit

SYNOPSIS

```
declare field_number        fixed binary(31);
declare field_data          char(256) varying;
declare occurrence          fixed binary(31);
declare modulus             fixed binary(31);
declare minimum_digits      fixed binary(31);
declare status              fixed binary(31);
status = xsm_ckdigit(field_number, field_data, occurrence,
         modulus, minimum_digits);
```

DESCRIPTION

This function is called by field validation. It verifies that `field_data` contains the required minimum number of digits terminated by the proper check digit. If not, it posts an error message before returning. It can also be used to check any character string or field. If `field_data` is null, the string to check is obtained from the `field_number` and `occurrence` and an error message is displayed if the string is bad. If `field_number` is zero, no message will be posted, but the function's return code will indicate whether the string passed its check.

A fuller description of `sm_ckdigit` is included with the source code, which is distributed with JAM.

Note that this function can be replaced by a user-installed check digit function which field validation will call instead. See the chapter on installing functions.

RETURNS

0 If the field contents are available and valid.
-1 If the field contents do not contain the minimum number of digits or the proper check digit.
-2 If the length of `field_data` is zero and the field or occurrence cannot be found

cl_all_mdts

clear all MDT bits

SYNOPSIS

```
call xsm_cl_all_mdts();
```

DESCRIPTION

Clears the MDT (modified data tag) of every occurrence, both onscreen and off.

JAM sets the MDT bit of an occurrence to indicate that it has been modified, either by keyboard entry or by a call to a function like `xsm_putfield`, since the screen was first displayed (i.e., after the screen entry function returns).

RELATED FUNCTIONS

```
field_number = xsm_tst_all_mdts(occurrence);
```

cl_unprot

clear all unprotected fields

SYNOPSIS

```
call xsm_cl_unprot();
```

DESCRIPTION

Erases onscreen and offscreen data from all fields that are not protected from clearing (CPROTECT). Date and time fields that take system values are re–initialized. Fields with the null edit are reset to their null indicator values.

This function is normally bound to the CLEAR ALL key.

RELATED FUNCTIONS

```
status = xsm_aprotect(field_number, mask);
```

clear_array

clear all data in an array

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_clear_array(field_number);

status = xsm_lclear_array(field_number);
```

DESCRIPTION

Both functions clear all data from the array containing the field specified by field_number. The value returned by `xsm_num_occurs` is changed to zero. The array is cleared even if it is protected from clearing (CPROTECT).

`xsm_clear_array` also clears arrays synchronized with the specified array, except for synchronized arrays that are protected from clearing.

`xsm_lclear_array` only clears the specified array.

RETURNS

-1 if the field does not exist;
0 otherwise.

VARIANTS

```
status = xsm_n_clear_array(field_name);
status = xsm_n_lclear_array(field_name);
```

RELATED FUNCTIONS

```
status = xsm_aprotect(field_number, mask);
status = xsm_protect(field_number);
```

close_window

close current window

SYNOPSIS

```
declare                    fixed binary(31);
status = xsm_close_window();
```

DESCRIPTION

xsm_close_window is used to close a window opened by xsm_r_window (or variant), xsm_r_at_cur (or variant), or xsm_mwindow.

The currently open window is erased, and the screen is restored to the state before the window was opened. All data from the window being closed is lost unless LDB processing is active, in which case named fields are copied to the LDB using xsm_lstore. Since windows are stacked, the effect of closing a window is to return to the previous window. The cursor reappears at the position it had before the window was opened.

When using the JAM Executive, use xsm_jclose to close a form. xsm_jclose will call xsm_jform to pop the form stack and open the new top form on the stack. In the case of a window, xsm_jclose will call xsm_close_window to close the window.

RETURNS

-1 is returned if there is no window open, (i.e. if the currently displayed screen is a form or if no screen is displayed).
0 is returned otherwise.

RELATED FUNCTIONS

```
status = xsm_r_window(screen_name, start_line, start_column);
return_value = xsm_wselect(window_number);
```

d_msg_line

display a message on the status line

SYNOPSIS

```
declare message           char(256) varying;
declare display_attribute fixed binary(31);
call xsm_d_msg_line(message, display_attribute);
```

DESCRIPTION

The message in `message` is displayed on the status line, with an initial display attribute of `display_attribute`. If the cursor position display has been turned on (see `xsm_c_vis`), the end of the status line will contain the cursor's current row and column. Messages displayed with `xsm_d_msg_line` override both background and field status text.

Messages posted with `xsm_d_msg_line` are displayed until the status line is cleared by `xsm_d_msg_line`. They will persist from screen to screen until cleared. Clearing is accomplished by passing `xsm_d_msg_line` an empty string for `message` and a 0 for `display_attribute`. Once cleared, any currently overidden message will resume. The function `xsm_d_msg_line` will itself be overridden by `xsm_err_reset` and related functions, or by the ready/wait message enabled by `xsm_setstatus`.

Possible values for `display_attribute` are defined in `smdefs.incl.pl1`, as shown in the table below:

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Highlights		Background Highlights	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Colors		Background Colors	
BLACK	0000	B_BLACK	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

Several *percent escapes* provide control over the content and presentation of status messages. The character following the percent sign must be in upper-case. Note that, if a message containing percent escapes is displayed before `xsm_initcrt` is called, the percent escapes will show up in the message.

If a string of the form `%A`nnnn appears anywhere in the message, the hexadecimal number nnnn is interpreted as a display attribute to be applied to the remainder of the message. The table gives the numeric values of the logical display attributes you will need to construct embedded attributes. If you want a digit to appear *immediately* after the attribute change, pad the attribute to 4 digits with leading zeros. If the following character is not a legal hex digit, then leading zeros are unnecessary.

If a string of the form `%K`keyname appears anywhere in the message, keyname is interpreted as a logical key mnemonic, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the `%K` is stripped out and the mnemonic remains. Key mnemonics are defined in `smkeys.incl.pl1`; it is of course the name, not the number, that you want here. The mnemonic must be in upper-case.

If the message begins with a `%B`, JAM will beep the terminal (using `xsm_bel`) before issuing the message.

RELATED FUNCTIONS

```
call xsm_err_reset(message);
call xsm_msg(column, disp_length, text);
status = xsm_mwindow(text, line, column);
```

dblval

get the value of a field as a real number

SYNOPSIS

```
declare field_number        fixed binary(31);
declare value               float binary(53);
value = xsm_dblval(field_number);
```

DESCRIPTION

This function returns the contents of `field_number` as a real number. It calls `xsm_strip_amt_ptr` to remove superfluous amount editing characters before converting the data.

RETURNS

The real value of the field is returned.
If the field is not found, the function returns 0.

VARIANTS

```
value = xsm_e_dblval(field_name, element);
value = xsm_i_dblval(field_name, occurrence);
value = xsm_n_dblval(field_name);
value = xsm_o_dblval(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_dtofield(field_number, value, format);
outbuf = xsm_strip_amt_ptr(field_number, inbuf, );
```

dd_able

turn LDB write–through on or off

SYNOPSIS

```
declare flag                fixed binary(31);
call xsm_dd_able(flag);
```

DESCRIPTION

During normal JAM processing, named fields in the screen and local data block are kept in sync. When a screen is displayed (and after the screen entry function completes), values are copied in from the LDB; when control passes from the screen (before the screen entry function is executed), values are copied back to the LDB. Normally, when application code reads or writes a value to or from a named field/LDB entry JAM treats the name as a field name unless no such field exists, in which case JAM treats the name as an LDB entry name. During screen entry and exit processing, this logic is reversed in order to preserve the illusion that screen and LDB entries that share the same name also share the same data.

`xsm_dd_able` turns this feature off if `flag` is "0" and on if it is "1". The feature is on by default. When it is off, the LDB is never accessed.

deselect

deselect a checklist occurrence

SYNOPSIS

```
declare group_name          char(256) varying;
declare group_occurrence    fixed binary(31);
declare status              fixed binary(31);
status = xsm_deselect(group_name, group_occurrence);
```

DESCRIPTION

This function allows you to deselect a specific occurrence within a checklist. The group name and occurrence number is used to reference the desired selection. See the Author's Guide for a more detailed discussion of groups.

Use `xsm_select` to select a group occurrence and `xsm_isselected` to check whether or not a particular group occurrence is currently selected.

NOTE: You can not deselect a radio button occurrence. Using `xsm_select` on a radio button occurrence will automatically deselect the current selection.

RETURNS

-1 arguments do not reference a checklist occurrence.
0 occurrence not previously selected.
1 occurrence previously selected.

RELATED FUNCTIONS

```
status = xsm_isselected(group_name, group_occurrence);
status = xsm_select(group_name, group_occurrence);
```

dicname

set data dictionary name

SYNOPSIS

```
declare dic_name          char(256) varying;
declare status            fixed binary(31);
status = xsm_dicname(dic_name);
```

DESCRIPTION

This function names the application's data dictionary, which is *data.dic* by default. It must be called before JAM initialization, in particular before `xsm_ldb_init` is called to initialize the local data block from the data dictionary. The argument `dic_name` is a character string giving the file name; JAM will search for it in all the directories in the `SMPATH` variable.

You can achieve the same effect by defining the `SMDICNAME` variable in your setup file equal to the data dictionary name. See the section on setup files in the Configuration Guide.

Use the function `xsm_pinquire` to find the name of the data dictionary in use.

RETURNS

–1 if it fails to allocate memory to store the name,
0 otherwise.

RELATED FUNCTIONS

```
buffer = xsm_pinquire(which);
```

disp_off

get displacement of cursor from start of field

SYNOPSIS

```
declare offset              fixed binary(31);
offset = xsm_disp_off();
```

DESCRIPTION

Returns the difference between the first column of the current field and the current cursor location. This function ignores offscreen data; use `xsm_sh_off` to obtain the total cursor offset of a shiftable field.

RETURNS

The difference between cursor position and start of field, or
–1 if the cursor is not in a field.

RELATED FUNCTIONS

```
call field_number = xsm_getcurno();
call offset = xsm_sh_off();
```

dlength

get the length of a field's contents

SYNOPSIS

```
declare field_number       fixed binary(31);
declare data_length        fixed binary(31);
data_length = xsm_dlength(field_number);
```

DESCRIPTION

Returns the length of data stored in `field_number`. The length does not include leading blanks in right justified fields, or trailing blanks in left–justified fields (which are also ignored by `xsm_getfield`). It does include data that have been shifted offscreen.

RETURNS

Length of field contents, or
–1 if the field is not found.

VARIANTS

```
data_length = xsm_e_dlength(field_name, element);
data_length = xsm_i_dlength(field_name, occurrence);
data_length = xsm_n_dlength(field_name);
data_length = xsm_o_dlength(field_number, occurrence);
```

RELATED FUNCTIONS

```
field_length = xsm_length(field_number);
```

do_region

rewrite part or all of a screen line

SYNOPSIS

```
declare line              fixed binary(31);
declare column            fixed binary(31);
declare length            fixed binary(31);
declare display_attribute fixed binary(31);
declare text              char(256) varying;
call xsm_do_region(line, column, length, display_attribute,
        text);
```

DESCRIPTION

The screen region defined by `line`, `column`, and `length` is rewritten. `Line` and `column` are counted *from zero*, with (0, 0) the upper left-hand corner of the screen.

If `text` is zero, the screen region is redrawn with whatever `display_attribute` has been assigned. If `text` is shorter than `length`, it is padded out with blanks. In either case, the display attribute of the whole area is changed to `display_attribute`.

Possible values for `display_attribute` are defined in `smdefs.incl.pl1`, as shown in the table below:

Foreground Attributes	*Background Attributes*
BLANK	B_HILIGHT
REVERSE	
UNDERLN	
BLINK	
HILIGHT	
STANDOUT	
DIM	
ACS (alternate character set)	

Foreground Colors	*Background Colors*
BLACK	B_BLACK
BLUE	B_BLUE
GREEN	B_GREEN
CYAN	B_CYAN
RED	B_RED
MAGENTA	B_MAGENTA
YELLOW	B_YELLOW
WHITE	B_WHITE

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

doccur

delete occurrences

SYNOPSIS

```
declare field_number       fixed binary(31);
declare occurrence         fixed binary(31);
declare count              fixed binary(31);
declare return_value       fixed binary(31);
return_value = xsm_o_doccur(field_number, occurrence, count);
```

DESCRIPTION

NOTE: This function only exists in the `o_` and `i_` variations. There is NO `xsm_doccur` since this function only applies to arrays.

This function deletes the data in `count` occurrences beginning with the specified `occurrence`. If the array is scrollable, then it deallocates `count` occurrences. The data in occurrences following the last deleted occurrence are moved up in the array so that there are no gaps. Fewer than `count` occurrences will be deleted if the number of remaining allocated occurrences, starting with the referenced occurrence, is less than `count`.

If `count` is negative, occurrences are inserted instead, subject to limitations explained at `xsm_ioccur`. The function `xsm_ioccur` is normally used to add blank occurrences.

If `occurrence` is zero, the occurrence used is that of `field_number`. If `occurrence` is nonzero, however, it is taken relative to the first field of the array in which `field_number` occurs.

Any clearing–unprotected synchronized arrays will have the same operations performed on them as the referenced array.

This function is normally bound to the DELETE LINE key.

RETURNS

–1 if the field or occurrence number was out of range;
–3 if insufficient memory was available;
otherwise, the number of occurrences actually deleted (zero or more).

VARIANTS

```
return_value = xsm_i_doccur(field_name, occurrence, count);
```

dtofield

write a real number to a field

SYNOPSIS

```
declare field_number        fixed binary(31);
declare value               float binary(53);
declare format              char(256) varying;
declare status              fixed binary(31);
status = xsm_dtofield(field_number, value, format);
```

DESCRIPTION

The real number `value` is converted to human–readable form, according to `format`, and moved into `field_number` via a call to `xsm_amt_format`. If the `format` string is empty, the number of decimal places will be taken from a data type edit, if one exists; failing that, from a currency edit, if one exists; or failing that, will default to 2.

The number of decimal places may be forced to be an arbitrary number n, via rounding, by using the format string `%.nf`". The format string `%t.nf`" may be used to truncate instead of to round.

RETURNS

–1 is returned if the field is not found.
–2 is returned if the output would be too wide for the destination field.
0 is returned otherwise.

VARIANTS

```
status = xsm_e_dtofield(field_name, element, value, format);
status = xsm_i_dtofield(field_name, occurrence, value, format);
status = xsm_n_dtofield(field_name, value, format);
status = xsm_o_dtofield(field_number, occurrence, value,
         format);
```

RELATED FUNCTIONS

```
status = xsm_amt_format(field_number, buffer);
value = xsm_dblval(field_number);
```

e_
variants that take a field name and element number

SYNOPSIS

```
declare field_name          char(256) varying;
declare element             fixed binary(31);
call xsm_e_...(field_name, element, ...);
```

DESCRIPTION

The `e_` variant functions access one element of an array by field name and element number. For a description of any particular function, look under the related function without `e_` in its name. For example, `xsm_e_amt_format` is described under `xsm_amt_format`.

Despite the fact that they take a field name as argument, these functions do not search the LDB for names not found in the screen because an element number is ambiguous when referring to the LDB.

edit_ptr

get special edit string

SYNOPSIS

```
declare buffer              char(256) varying;
declare field_number        fixed binary(31);
declare edit_type           fixed binary(31);
buffer = xsm_edit_ptr(field_number, edit_type);
```

DESCRIPTION

This function searches the special edits area of a field or group for an edit of type `edit_type`. The `edit_type` should be one of the following values, which are defined in `smdefs.incl.pl1`:

Edit type	*Contents of edit string*
NAMED	Field name
CPROG	Name of field validation function
FE_CPROG	Name of field entry function
FX_CPROG	Name of field exit function
HELPSCR	Name of help screen
HARDHLP	Name of automatic help screen
HARDITM	Name of automatic item selection screen
ITEMSCR	Name of item selection screen
SUBMENU	Name of pull-down menu screen
TABLOOK	Name of screen for table-lookup validation
NEXTFLD	Next field (contains both primary and alternate fields)
PREVFLD	Previous field (contains both primary and alternate fields)
TEXT	Status line prompt

Edit type	*Contents of edit string*
MEMO1 ... MEMO9	Nine arbitrary user-supplied text strings
JPLTEXT	Attached JPL code
CALC	Math expression executed at field exit
CKDIGIT	Flag and parameters for check digit
FTYPE	Data type for inclusion in structure
RETCODE	Return value for menu or return entry field
CMASK	Regular expression for field validation
CCMASK	Regular expression for character validation
CKBOX	Offset and attribute of checkbox in a group
ALTSC_CPROG	Name of alternate scrolling function
KEYSET	Name of keyset associated with screen.
SDATETIME	Date/time field with user format, initialized with system values.
UDATETIME	Date/time field with user format, initialized by the user.
CURRED	Currency field format, see `smdefs.incl.pl1` for details.
NULLFIELD	Null field representation.
RANGEL	Low bound on range; up to 9 permitted
RANGEH	High bound on range; up to 9 permitted
EDT_BITS	Normally for internal use (see `smdefs.incl.pl1` for more information.)

The string returned by `xsm_edit_ptr` contains:

- The total length of the string (including the two overhead bytes and any terminators) in its first byte.

- The edit_type code in its second byte.
- The body of the edit in the subsequent bytes. Refer to the source listing for the file `smdefs.incl.pll` for specific information on how to interpret each individual edit.

If the field has no edit of type `edit_type`, the returned buffer will contain a zero. If a field has multiple edits of one type, such as RANGEH or RANGEL, then each additional edit is added onto the end of the string following the same pattern as the first one. For example, the first byte would contain the length of the string up to the end of the body of the edit of RANGEH. Adding one to this number would give you the byte that contains the length of the string containing information on RANGEL and so forth.

This function is especially useful for retrieving user-defined information contained in MEMO edits.

In the case of groups, the edits `PREVFLD`, `NEXTFLD`, `CPROG`, `FE_CPROG`, and `FE_CPROG` may be used to obtain group information.

RETURNS

The first (length) byte of the special edit of the field.
0 if the field or edit is not found.

VARIANTS

```
buffer = xsm_n_edit_ptr(field_name, edit_type);
```

emsg

display an error message and reset the message line without turning on the cursor

SYNOPSIS

```
declare message          char(256) varying;
call xsm_emsg(message);
```

DESCRIPTION

This function displays `message` on the status line, if it fits, or in a window if it is too long. If the cursor position display has been turned on (see `xsm_c_vis`), the end of the status line will contain the cursor's current row and column. If the message text would overlap that area of the status line, it will be displayed in a window instead. The message remains visible until the operator presses a key. The function's exact behavior in dismissing the message is subject to the error message options; see `xsm_option`.

`xsm_emsg` is identical to `xsm_err_reset`, except that it does not attempt to turn the cursor on before displaying the message. It is similar to `xsm_qui_msg`, which inserts a constant string (normally "ERROR:") before the message.

Several *percent escapes* provide control over the content and presentation of status messages. The character following the percent sign must be in upper-case. Note that, if a message containing percent escapes is displayed before `xsm_initcrt` is called, the percent escapes will show up in the message.

If a string of the form `%A`nnnn appears anywhere in the message, the hexadecimal number nnnn is interpreted as a display attribute to be applied to the remainder of the message. The table gives the numeric values of the logical display attributes you will need to construct embedded attributes. If you want a digit to appear *immediately* after the attribute change, pad the attribute to 4 digits with leading zeros. If the following character is not a legal hex digit, then leading zeros are unnecessary.

If a string of the form `%K`keyname appears anywhere in the message, keyname is interpreted as a logical key mnemonic, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the `%K` is stripped out and the mnemonic remains. Key mnemonics are defined in `smkeys.incl.pl1`; it is of course the name, not the number, that you want here. The mnemonic must be in upper-case.

If the message begins with a `%B`, JAM will beep the terminal (using `xsm_bel`) before issuing the message.

If `%N` appears anywhere in the message, the latter will be presented in a pop–up window rather than on the status line, and all occurrences of `%N` will be replaced by new lines.

If the message begins with `%W`, it will be presented in a pop–up window instead of on the status line. The window will appear near the bottom center of the screen, unless it would obscure the current field by so doing; in that case, it will appear near the top.

If the message begins with `%Mu` or `%Md`, JAM will ignore the default error message acknowledgement flag and process (for `%Mu`) or discard (for `%Md`) the next character typed.

Possible hex values for display attribute are defined in `smdefs.incl.pll`, as shown in the table below:

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Highlights		Background Highlights	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Colors		Background Colors	
BLACK	0000	B_BLACK	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

RELATED FUNCTIONS

```
call xsm_err_reset(message);
call xsm_qui_msg(message);
call xsm_quiet_err(message);
```

err_reset

display an error message and reset the status line

SYNOPSIS

```
declare message          char(256) varying;
call xsm_err_reset(message);
```

DESCRIPTION

The `message` is displayed on the status line until acknowledged it by pressing a key. If `message` is too long to fit on the status line, it is displayed in a window instead. If the cursor position display has been turned on (see `xsm_c_vis`), the end of the status line will contain the cursor's current row and column. If the message text would overlap that area of the status line, it will be displayed in a window instead. The exact behavior of error message acknowledgement is governed by `xsm_option`. The initial message attribute is set by `xsm_option`, and defaults to blinking.

This function turns the cursor on before displaying the message, and forces off the global flag `sm_do_not_display`. It is similar to `xsm_emsg`, which does not turn on the cursor, and to `xsm_quiet_err`, which inserts a constant string (normally "ERROR:") before the message.

Several *percent escapes* provide control over the content and presentation of status messages. See `xsm_emsg` for details.

RELATED FUNCTIONS

```
call xsm_emsg(message);
call xsm_qui_msg(message);
call xsm_quiet_err(message);
```

fi_path

return the full path name of a file

SYNOPSIS

```
declare buffer              char(256) varying;
declare file_name           char(256) varying;
buffer = xsm_fi_path(file_name);
```

DESCRIPTION

Use this function to find the full path name of a file. The file may be a screen or any other type of file. The file's full path name is returned in `buffer`.

The file name is first sought in the current directory. If that fails, the path given to `xsm_initcrt` is checked. Finally the path defined by `SMPATH` is searched.

RETURNS

0 if the file cannot be found in any path.
Else, The path is returned in `buffer`.

finquire

obtain information about a field

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare field_number       fixed binary(31);
declare which              fixed binary(31);
declare value              fixed binary(31);
value = xsm_finquire(field_number, which);
```

DESCRIPTION

Use this function to obtain various information about a field. The variable `which` is a mnemonic that specifies the particular piece of information desired.

Mnemonics for `which` are defined in the file `smglobs.incl.pl1`. The following values are available:

Mnemonic	*Meaning*
`FD_LINE`	Line that field is on.
`FD_COLM`	Column of field's first position.
`FD_ATTR`	Field attributes (see `smdefs.incl.pl1`).
`FD_LENG`	Onscreen field length.
`FD_ASIZE`	Onscreen array size (1 if scalar).
`FD_ELT`	Onscreen element number.
`FD_SHLENG`	Shiftable length.
`FD_SHINCR`	Shift increment.
`FD_SHOFS`	Current shift offset (number of positions field has been shifted; 0 if shifted to left edge).
`FD_SCINCR`	Scrolling increment (for Next/Prev page keys).
`FD_SCFLAG`	Scrolling array circular? (T/F).

Mnemonic	*Meaning*
FD_SCATTR	Scrolling occurrence display attributes set with xsm_i_achg; zero if onscreen element attributes is to be used. For xsm_i_finquire variant only.
FD_FELT	First onscreen occurrence of scrolling array (1 if scrolled to top).

RETURNS

The value of which if found.
0 otherwise.

VARIANTS

```
value = xsm_e_finquire(field_name, element, which);
value = xsm_i_finquire(field_name, occurrence, which);
value = xsm_n_finquire(field_name, which);
value = xsm_o_finquire(field_number, occurrence, which);
```

RELATED FUNCTIONS

```
value = xsm_gp_inquire(group_name, which);
value = xsm_inquire(which);
value = xsm_iset(which, newval);
buffer = xsm_pinquire(which);
buffer = xsm_pset(which, newval);
```

fldno

get the field number of an array element or occurrence

SYNOPSIS

```
declare field_name          char(256) varying;
declare field_number        fixed binary(31);
field_number = xsm_n_fldno(field_name);
```

DESCRIPTION

NOTE: This function only exists in the `e_`, `i_`, `n_`, and `o_` variations. There is NO `xsm_fldno` since this function determines the field number given other information.

The `e_` variant returns the field number of an array element specified by `field_name` and `element`. If `element` is zero, then `xsm_e_fldno` returns the field number of the named field, or the base element of the named array.

The `i_` and `o_` variants return the number of the field containing the specified occurrence if the occurrence is onscreen, or 0 if the occurrence is offscreen.

The `n_` variant returns the field number of a field specified by name, or the base field number of an array specified by name.

RETURNS

0 if the name is not found, if the element number exceeds 1 and the named field is not an array, or if the occurrence is offscreen.
Otherwise, returns an integer between 1 and the maximum number of fields on the current screen that represents the field number.

VARIANTS

```
field_number = xsm_e_fldno(field_name, element);
field_number = xsm_i_fldno(field_name, occurrence);
field_number = xsm_o_fldno(field_number, occurrence);
```

flush

flush delayed writes to the display

SYNOPSIS

```
call xsm_flush();
```

DESCRIPTION

This function performs delayed writes and flushes all buffered output to the display. It is called automatically via `xsm_input` whenever the keyboard is opened and there are no keystrokes available, *i.e.* typed ahead.

Calling this routine indiscriminately can significantly slow execution. As it is called whenever the keyboard is opened, the display is always guaranteed to be in sync before data entry occurs; however, if you want timed output or other non-interactive display, use of this routine will be necessary.

RELATED FUNCTIONS

```
call xsm_flush();
call xsm_rescreen();
```

form

display a screen as a form

SYNOPSIS

```
declare screen_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_r_form(screen_name);

declare screen_address     bit(0);
declare status             fixed binary(31);
status = xsm_d_form(screen_address);

declare lib_desc           fixed binary(31);
declare screen_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_l_form(lib_desc, screen_name);
```

DESCRIPTION

This set of functions is primarily intended to be used by developers who are writing their own executive. These functions do not update the form stack, so it is generally not a good idea to use them with the JAM Executive. To open a form while under the control of the JAM Executive, use a JAM control string or `xsm_jform`.

These functions display the named screen as a base form. Bringing up a screen as a form with `xsm_d_form`, `xsm_l_form`, `xsm_r_form` causes the previously displayed form and windows to be discarded, and their memory freed. The new screen is displayed with its upper left-hand corner at the extreme upper left of the display (position (0, 0)).

If an error occurs a return of –1 or –2 means that the previously displayed form is still displayed and may be used. Other negative return codes indicate that the display is undefined. The caller should display another form before using Screen Manager functions.

When you use `xsm_r_form` the named screen is sought first in the memory-resident screen list, and if found there is displayed using `xsm_d_form`. It is next sought in all the open screen libraries, and if found is displayed using `xsm_l_form`. Next it is sought on disk in the current directory; then under the path supplied to `xsm_initcrt`; then in all the paths in the setup variable SMPATH. If any path exceeds 80 characters, it is skipped. If the entire search fails, this function displays an error message and returns.

You may save processing time by using `xsm_d_form` to display screens that are memory-resident. Use `bin2pll` to convert screens from disk files, which you can modify us-

ing `jxform`, to program data structures you can compile into your application. A memory–resident screen is never altered at run–time, and may therefore be made shareable on systems that provide for sharing read–only data. `xsm_r_form` can also display memory–resident screens, if they are properly installed using `xsm_formlist`. Memory–resident screens are particularly useful in applications that have a limited number of screens, or in environments that have a slow disk (*e g*. MS–DOS). `screen_address` is the address of the screen in memory.

You may also save processing time by using `xsm_l_form` to display screens that are in a library. A library is a single file containing many screens (and/or JPL modules and keysets). You can assemble one from individual screen files using the utility `formlib`. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on the number of paths searched.

The library descriptor, `lib_desc`, is an integer returned by `xsm_l_open`, which you must call before trying to read any screens from a library. Note that `xsm_r_form` also searches any open libraries.

To display a window use `xsm_r_at_cur`, `xsm_r_window`, or one of their variants.

RETURNS

0 if no error occurred
–1 if the screen file's format is incorrect; previous form still displayed and available
–2 if the screen cannot be found or the maximum allowable number of files is already open; previous form still displayed and available
–4 if, after the screen has been cleared, the screen cannot be successfully displayed because of a read error;
–5 if, after the screen was cleared, the system ran out of memory;

RELATED FUNCTIONS

```
status = xsm_r_window(screen_name, start_line, start_column);
status = xsm_r_at_cur(screen_name);
```

formlist

update list of memory–resident files

SYNOPSIS

```
declare name                char(256) varying;
declare address             bit(0);
declare status              fixed binary(31);
status = xsm_formlist(name, address);
```

DESCRIPTION

This function adds a JPL module, keyset, or screen to the memory resident form list. Each member of the list is a structure giving the name of the JPL module, screen, or keyset, as a character string, and its address in memory. This function is commonly called from `main`. It can be called any number of times from an application program to augment to the memory resident list.

The library functions `xsm_r_form`, `xsm_r_window`, `xsm_r_at_cur`, and `xsm_r_keyset` all take a screen or keyset name as a parameter and search for it in the memory–resident list before attempting to read the screen or keyset from disk. The `jpl` command (see the JPL Programmer's Guide) and the function `xsm_jplcall` search the memory resident form list when looking for a JPL procedure to execute.

To make a JPL module, keyset, or screen memory resident, you can use the `bin2pl1` utility to create a static PL/1 structure initialized with the binary content of the object. You must then compile and link the structure with the application executable.

RETURNS

–1 if insufficient memory is available for the new list;
0 otherwise.

RELATED FUNCTIONS

```
call xsm_rmformlist;
```

fptr
get the content of a field

SYNOPSIS

```
declare field_number        fixed binary(31);
declare buffer              char(256) varying;
buffer = xsm_fptr(field_number);
```

DESCRIPTION

This routine returns the contents of the field specified by `field_number`. Leading blanks in right–justified fields and trailing blanks in left–justified fields are stripped.

RETURNS

The field contents, or
0 if the field cannot be found.

VARIANTS

```
buffer = xsm_e_fptr(field_name, element);
buffer = xsm_i_fptr(field_name, occurrence);
buffer = xsm_n_fptr(field_name);
buffer = xsm_o_fptr(field_number, occurrence);
```

RELATED FUNCTIONS

```
length = xsm_getfield(buffer, field_number);
status = xsm_putfield(field_number, data);
```

ftog

convert field references to group references

SYNOPSIS

```
declare field_number       fixed binary(31);
declare group_occurrence   fixed binary(31);
declare buffer             char(256) varying
buffer = xsm_ftog(field_number, group_occurrence);
```

DESCRIPTION

This function converts field references to group references. Use `xsm_i_gtof` to convert them back.

This function returns the name of the group containing the referenced field and inserts its group occurrence number into the address of `occurrence`.

RETURNS

The group name if found and indirectly through `group_occurrence` the group occurrence number.
0 otherwise and `group_occurrence` is unchanged.

VARIANTS

```
buffer = xsm_e_ftog(field_name, element, group_occurrence);
buffer = xsm_i_ftog(field_name, occurrence, group_occurrence);
buffer = xsm_n_ftog(field_name, group_occurrence);
buffer = xsm_o_ftog(field_number, occurrence,
         group_occurrence);
```

RELATED FUNCTIONS

```
field_number = xsm_i_gtof(group_name, group_occurrence,
         occurrence);
```

ftype

get the data type and precision of a field

SYNOPSIS

```
declare field_number        fixed binary(31);
declare precision_ptr       fixed binary(31);
declare type                fixed binary(31);
type = xsm_ftype(field_number, precision_ptr);
```

DESCRIPTION

This function analyzes the edits of a field or LDB entry, and returns data type information. First the "type" (FTYPE) edit is checked, then the "currency" edit, the "date/time" edit, and finally the "character" edit.

Note that this differs from the functionality of xsm_rdstruct, xsm_wrtstuct, xsm_rrecord, and xsm_wrecord. These functions only test the type and character edits. They use the currency edit only to determine the precision of a numeric field that has no type edit.

This function returns an integer containing the data type code, plus any applicable flags. The data type codes and flags are detailed in the tables below.

Data Type Code	*Meaning*
FT_CHAR	Type edit is *char string*; or character edit is *unfiltered, letters only, alphanumeric,* or *regular expression*
FT_INT	Type edit is *int*
FT_UNSIGNED	Type edit is *unsigned int*; or character edit is *digit*
FT_SHORT	Type edit is *short int*
FT_LONG	Type edit is *long int*
FT_FLOAT	Type edit is *float*
FT_DOUBLE	Type edit is *double*; or character edit is *numeric*
FT_ZONED	Type edit is *zoned dec.*
FT_PACKED	Type edit is *packed dec.*

Data Type Code	*Meaning*
DT_YESNO	Character edit is *yes/no*
DT_CURRENCY	Currency edit
DT_DATETIME	Date/time edit

Flag	*Meaning*
DF_NULL	Null edit
DF_REQUIRED	Data required edit (not applicable to LDB)
DF_WRAP	Word wrap edit
DF_OMIT	Type edit is *omit*.

To determine the data type code, check this integer for each flag in the fashion of the example field function shown on page 14, starting with DF_OMIT and working up the list.. The value remaining will be the data type code.

Note that FT_OMIT is not listed as one of the data types. A field that has the type edit *omit* will return the data type determined by any of the other edits, as well as a flag indicating that it has the *omit* type edit.

The function will put the precision of float, double and currency values in the precison_ptr argument.

RETURNS

major data type code plus any applicable flags (see tables above).
0 if field is not found

VARIANTS

```
type = xsm_n_ftype(field_number, precision_ptr);
```

fval

force field validation

SYNOPSIS

```
declare field_number        fixed binary(31);
declare status              fixed binary(31);
status = xsm_fval(field_number);
```

DESCRIPTION

This function performs all validations on the indicated field or occurrence, and returns the result. If the field is protected against validation, the checks are not performed and the function returns 0; see `xsm_aprotect`. Validations are done in the order listed below. Some will be skipped if the field is empty, or if its VALIDED bit is already set (implying that it has already passed validation).

Validation	*Skip if valid*	*Skip if empty*
required	y	n
must fill	y	y
regular expression	y	y
range	y	y
check–digit	y	y
date or time	y	y
table lookup	y	y
currency format	y	n*
math expresssion	n	n
field validation	n	n
JPL function	n	n

* The currency format edit contains a skip–if–empty flag; see the Author's Guide.

If you need to force a skip–if–empty validation, make the field required. A field with embedded punctuation must contain at least one non–blank non–punctuation character in or-

der to be considered non–empty; otherwise any non blank character makes the field non–empty.

Math expressions, JPL functions and field validation functions are never skipped, since they can alter fields other than the one being validated.

Field validation is performed automatically within `xsm_input` when the cursor exits a field via the `TAB` or `NL` logical keys. All fields on a screen are validated when `XMIT` is pressed (see `xsm_s_val`). Application programs need call this function only to force validation of other fields.

RETURNS

–2 if the field or occurrence specification is invalid;
–1 if the field fails any validation;
0 otherwise.

VARIANTS

```
status = xsm_e_fval(array_name, element);
status = xsm_i_fval(field_name, occurrence);
status = xsm_n_fval(field_name);
status = xsm_o_fval(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_n_gval(group_name);
status = xsm_s_val();
```

getcurno

get current field number

SYNOPSIS

```
declare field_number       fixed binary(31);
field_number = xsm_getcurno();
```

DESCRIPTION

This function returns the number of the field in which the cursor is currently positioned. The field number ranges from `1` to the total number of fields in the screen.

RETURNS

Number of the current field, or
0 if the cursor is not within a field.

RELATED FUNCTIONS

```
occurrence = xsm_occur_no();
```

getfield

copy the contents of a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare length             fixed binary(31);
length = xsm_getfield(buffer, field_number);
```

DESCRIPTION

This function copies the data found in `field_number` to `buffer`. Leading blanks in right-justified fields and trailing blanks in left-justified fields are not copied. The variants that reference a field by name will attempt to get data from the corresponding LDB entry if there is no such field on the screen (except that the order is reversed during screen entry/exit processing).

Responsibility for providing a buffer large enough for the field's contents rests with the calling program. This should be at least one greater than the maximum length of the field, taking shifting into account.

In variants that take `name` as an argument, either the name of a field or a group may be used. In the case of groups, `xsm_isselected` is preferred to `xsm_getfield` for determining whether or not a group occurrence is selected. If `xsm_n_getfield` is called on a radio button, the value in `buffer` will be the occurrence number of the selected item. If `xsm_i_getfield` is called on a checklist, the value in the first occurrence of the array will be the number of the first selected item in the group, the value in the second occurrence will be the number of the next selected item in the group and so on. If a checklist has, for example, three items selected, the fourth array occurrence will be empty.

Note that the order of arguments to this function is different from that to the related function `xsm_putfield`.

RETURNS

The total length of the field's contents, or
-1 if the field cannot be found.

VARIANTS

```
length = xsm_e_getfield(buffer, name, element);
length = xsm_i_getfield(buffer, name, occurrence);
length = xsm_n_getfield(buffer, name);
length = xsm_o_getfield(buffer, field_number, occurrence);
```

RELATED FUNCTIONS

```
buffer = xsm_fptr(field_number);
status = xsm_isselected(group_name, group_occurrence);
status = xsm_putfield(field_number, data);
```

getjctrl

get control string associated with a key

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare key                 fixed binary(31);
declare default             fixed binary(31);
declare buffer              char(256) varying
buffer = xsm_getjctrl(key, default);
```

DESCRIPTION

Each JAM screen contains a table of control strings associated with function keys. JAM also maintains a default table of keys and control strings, which take effect when the current screen has no control string for a function key you press. This table enables you to define system–wide actions for keys. It is initialized from SMINICTRL setup variables. See the section on setup in the Configuration Guide for further information.

This function searches one of the tables for `key`, a logical key mnemonic found in `smkeys.incl.pl1`, and returns a the associated control string. If `default` is zero, the table for the current screen is searched; otherwise, the system–wide table is searched.

RETURNS

The control string
0 if none is found.

RELATED FUNCTIONS

```
status = xsm_putjctrl(key, control_string, default);
```

getkey

get logical value of the key hit

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare key                fixed binary(31);
key = xsm_getkey();
```

DESCRIPTION

This function gets and interprets keyboard input and returns the logical value to the calling program. Normal characters are returned unchanged. Logical keys are interpreted according to a key translation file for the particular terminal you are using. See the Keyboard Input section in this guide, the Key Translation section in the Configuration Guide, and the modkey section in the Utilities Guide. `xsm_getkey` is normally not needed for application programming, since it is called by `xsm_input`.

Logical keys include TRANSMIT, EXIT, HELP, LOCAL PRINT, arrows, data modification keys like INSERT and DELETE CHAR, user function keys PF1 through PF24, shifted function keys SPF1 through SPF24, and others. Defined values for all are in `smkeys.incl.pl1`. A few logical keys, such as LOCAL PRINT and RESCREEN, are processed locally in `xsm_getkey` and not returned to the caller.

There is another function called `xsm_ungetkey`, which pushes logical key values back on the input stream for retrieval by `xsm_getkey`. Since all JAM input routines call `xsm_getkey`, you can use it to generate any input sequence automatically. When you use it, calls to `xsm_getkey` will not cause the display to be flushed, as they do when keys are read from the keyboard.

There are a number of user–installed functions that may be called by `xsm_getkey`. For further information see the section on installing functions in the Programmer's Guide.

Finally, there is a mechanism for detecting an externally established abort condition, essentially a flag, which causes JAM input functions to return to their callers immediately. The present function checks for that condition on each iteration, and returns the ABORT key if it is true. See `xsm_isabort`.

Application programmers should be aware that JAM control strings are not executed within this function, but at a higher level within the JAM run–time system (i.e., functions that call `xsm_getkey`. If you call this function, do not expect function key control strings to work.

The multiplicity of calls to user functions in `xsm_getkey` makes it a little difficult to see how they interact, which take precedence, and so forth. In an effort to clarify the process, we present an outline of `xsm_getkey`. The process of key translation is deliberately omitted, for the sake of clarity; that algorithm is presented separately, in the keyboard translation section of the Programmer's Guide.

***Step 1

- If an abort condition exists, return the ABORT key.
- If there is a key pushed back by ungetkey, return that.
- If playback is active and a key is available, take it directly to Step 2; otherwise read and translate input from the keyboard. When the keyboard is read, then the asynchronous function (if one is installed) is called during periods of keyboard inactivity.

*** Step 2

- Pass the key to the keychange function. If that function says to discard the key, go back to Step 1; otherwise if an abort condition exists, return the ABORT key.
- If recording is active, pass the key to the recording function.

*** Step 3

- If the routing table says the key is to be processed locally, do so.
- If the routing table says to return the key, return it; otherwise, go back to Step 1.
- If the key is a soft key, return its logical value.

RETURNS

The standard ASCII value of a displayable key; a value greater than 255 (FF hex) for a key sequence in the key translation file.

RELATED FUNCTIONS

```
old_flag = xsm_keyfilter(flag);
return_value = xsm_ungetkey(key);
```

gofield

move the cursor into a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_gofield(field_number);
```

DESCRIPTION

Positions the cursor to the first enterable position of `field_number`. If the field is shiftable, it is reset.

In a right–justified field, the cursor is placed in the rightmost position and in a left–justified field, in the leftmost. In either case, if the field has embedded punctuation, the cursor goes to the nearest position not occupied by a punctuation character. Use `xsm_off_gofield` to place the cursor in position other than that of the first character of a field.

When called to position the cursor in a scrollable array, `xsm_o_gofield` and `xsm_i_gofield` return an error if the occurrence number passed exceeds by more than 1 the number of allocated occurrences in the specified array. If the desired occurrence is offscreen, it is scrolled on–screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
status = xsm_e_gofield(field_name, element);
status = xsm_i_gofield(field_name, occurrence);
status = xsm_n_gofield(field_name);
status = xsm_o_gofield(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_off_gofield(field_number, offset);
```

gp_inquire

obtain information about a group

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare group_name          char(256) varying;
declare which               fixed binary(31);
declare value               fixed binary(31);
value = xsm_gp_inquire(group_name, which);
```

DESCRIPTION

Use this function to obtain various information about group. The variable `which` is a mnemonic that specifies the particular piece of information desired.

Mnemonics for `which` are defined in the file `smglobs.incl.pl1`. They are:

Mnemonic	*Meaning*
GP_NOCCS	Number of occurrences in the group (sum of number of occurrences of all fields/arrays in group)
GP_FLAGS	Flags

RETURNS

The value of `which`, if found, or
-1 otherwise.

gtof

convert a group name and index into a field number and occurrence

SYNOPSIS

```
declare group_name          char(256) varying;
declare group_occurrence    fixed binary(31);
declare occurrence          fixed binary(31);
declare field_number        fixed binary(31);
field_number = xsm_i_gtof(group_name, group_occurrence,
        occurrence);
```

DESCRIPTION

NOTE: This function only exists in the `i_` variation. There is no `xsm_gtof` since groups cannot be referenced by number.

Use this function to convert a group name and group_occurrence into a field number and occurrence. The variable `group_name` is the name of the group and `group_occurrence` is the specific field within the group.

The function returns the field number of the referenced field and inserts the occurrence number into the memory location addressed by `occurrence`.

Using this function allows you to use other JAM library routines to manipulate group fields by converting group references into field references. For instance, if you wanted to access text from a specific field within a group you would need to use `xsm_i_gtof` to get the field and occurrence number before you could use the function `xsm_o_getfield` to retrieve the text.

RETURNS

The field number if found.
0 otherwise.

RELATED FUNCTIONS

```
buffer = xsm_ftog(field_number, group_occurrence);
```

gval
force group validation

SYNOPSIS

```
declare group_name          char(256) varying;
declare status              fixed binary(31);
status = xsm_n_gval(group_name);
```

DESCRIPTION

NOTE: This function only exists in the `xsm_n_gval` variation. There is no `xsm_gval` since groups cannot be referenced by number.

Use this function to force the execution of a group's validation function. Use `xsm_s_val` to validate all fields and groups on the screen.

RETURNS

-1 if the group fails any validation.
-2 if the group name is invalid.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_fval(field_number);
status = xsm_s_val();
```

gwrap

get the contents of a wordwrap array

SYNOPSIS

```
declare buffer              char(256) varying;
declare field_number        fixed binary(31);
declare buffer_length       fixed binary(31);
declare length              fixed binary(31);
length = xsm_gwrap(buffer, field_number, buffer_length);
```

DESCRIPTION

This function copies the contents of the array specified by `field_number`, one occurrence at a time, into `buffer`, up to the size specified by `buffer_length`. A space is inserted before every non–empty occurrence, except the first.

The variant `xsm_o_gwrap` copies the contents of the array, beginning with the specified occurrence.

RETURNS

The length of transferrable data. If this is greater than `buffer_length`, then the data was truncated.

–1 if the field number is invalid or `buffer_length` is ≤ 0.

VARIANTS

```
status = xsm_o_gwrap(buffer, field_number, occurrence,
         buffer_length);
```

RELATED FUNCTIONS

```
status = xsm_pwrap(field_number, text);
```

hlp_by_name
display help window

SYNOPSIS

```
declare help_screen        char(256) varying;
declare status             fixed binary(31);
status = xsm_hlp_by_name(help_screen);
```

DESCRIPTION

The named screen is displayed and processed as a normal help screen, including input processing for the current field (if any).

Refer to the Author's Guide for instructions on how to create various kinds of help screens and for details of the behaviour of help screens.

RETURNS

-1 if screen is not found or other error;
1 if data copied from help screen to underlying field;
0 otherwise.

home

home the cursor

SYNOPSIS

```
declare field_number       fixed binary(31);
field_number = xsm_home();
```

DESCRIPTION

This function moves the cursor to the first enterable position of the first tab–unprotected field on the screen. If the screen has no tab–unprotected fields, the cursor is moved to the first line and column of the topmost screen. However, if you are using the JAM Executive, the cursor may not be visible if there are no tab–unprotected fields.

The cursor will be put into a tab–protected field if it occupies the first line and column of the screen and there are no tab–unprotected fields.

This function doesn't immediately trigger field entry, exit, or validation processing. Processing is based on the cursor position when control returns to `xsm_input`.

When the JAM logical key HOME is hit, `xsm_home` is called.

RETURNS

The number of the field in which the cursor is left, or
0 if the form has no unprotected fields and the home position is not in a protected field.

RELATED FUNCTIONS

```
call xsm_backtab();
status = xsm_gofield(field_number);
call xsm_last();
call xsm_nl();
call xsm_tab();
```

i_

variants that take a field name and occurrence number

SYNOPSIS

```
declare field_name          char(256) varying;
declare occurrence          fixed binary(31);
call xsm_i_...(field_name, occurrence, ...);
```

DESCRIPTION

The `i_` variants each refer to data by field name and occurrence number. An occurrence is a slot within an array in which data may be stored. Occurrences may be either on or off-screen. Since JAM treats an individual field as an array with one field, even a single non-scrolling field is considered to have one occurrence. The JAM library contains routines that allow you to manipulate individual occurrences during run-time.

If `occurrence` is zero, the reference is always to the current contents of the named field, or of the base field of the named array.

For the description of a particular function, look under the related function without `i_` in its name. For example, `xsm_i_amt_format` is described under `xsm_amt_format`.

If the named field is not part of the screen currently being displayed, these functions will attempt to retrieve or change its value in the local data block.

ininames

record names of initial data files for local data block

SYNOPSIS

```
declare name_list          char(256) varying;
declare status             fixed binary(31);
status = xsm_ininames(name_list);
```

DESCRIPTION

Use this routine to set up a list of initialization files for local data block entries. The file names in the single string `name_list` should be separated by commas, semicolons or blanks. There may be up to ten file names. You may achieve the same effect by defining the `SMININAMES` variable in your setup file to the list of names. See setup files in the Configuration Guide and the Data Dictionary chapter of the Author's Guide for details.

The files contain pairs of names and values, which are used to initialize local data block entries by `xsm_ldb_init`. This function is called during JAM initialization, so `xsm_ininames` should be called before then. White space in the initialization files is ignored, but we suggest a format like the following:

```
"emperor"      "Julius Caesar"
"lieutenant"   "Mark Antony"
"assassin[1]"  "Brutus"
"assassin[2]"  "Cassius"
```

Entries of all scopes may be freely mixed within all files. We recommend, however, that entries be grouped in files by scope if you are planning to use `xsm_lreset`. Use `xsm_lreset` to clear all entries of a given scope before reinitializing them from a single file.

RETURNS

–5 if insufficient memory is available to store the names;
0 otherwise.

RELATED FUNCTIONS

```
call xsm_ldb_init();
status = xsm_lreset(file_name, scope);
```

initcrt

initialize the display and **JAM** data structures

SYNOPSIS

```
declare path                char(256) varying;
call xsm_initcrt(path);

declare path                char(256) varying;
call xsm_jinitcrt(path);

declare path                char(256) varying;
call xsm_jxinitcrt(path);
```

DESCRIPTION

The function `xsm_initcrt` is intended for use only with a user–written executive. It is called automatically by the **JAM** Executive.

`xsm_initcrt` must be called at the beginning of screen handling, that is, before any screens are displayed or the keyboard opened for input to a **JAM** screen. Functions that set options, such as `xsm_option`, and those that install functions or configuration files such as `xsm_uinstall` or `xsm_vinit`, are the only kind that may be called before `xsm_initcrt`.

The argument `path` is a directory to be searched for screen files by `xsm_r_window` and variants. First the file is sought in the current directory; if it is not there, it is sought in the path supplied to this function. If it is not there either, the paths specified in the environment variable `SMPATH` (if any) are tried. The `path` argument *must* be supplied. If all forms are in the current directory, or if (as JYACC suggests) all the relevant paths are specified in `SMPATH`, an empty string may be passed. After setting up the search path, `xsm_initcrt` performs several initializations:

1. It calls a user–defined initialization routine.
2. It determines the terminal type, if possible by examining the environment (`TERM` or `SMTERM`), otherwise by asking the user.
3. It executes the setup files defined by the environment variables `SMVARS` and `SMSETUP`, and reads in the binary configuration files (message, key, and video) specific to the terminal.
4. It allocates memory for a number of data structures shared among **JAM** library functions.

5. If supported by the operating system, keyboard interrupts are trapped to a routine that clears the display and exits.
6. It initializes the operating system display and keyboard channels, and clears the display.

The functions `xsm_jinitcrt` and `xsm_jxinitcrt` are called by `jmain.pl1` and `jxmain.pl1` respectively for applications that use the JAM Executive. They, in turn, call `xsm_initcrt`.

RELATED FUNCTIONS

```
call xsm_resetcrt();
call xsm_jresetcrt();
call xsm_jxresetcrt();
```

input

open the keyboard for data entry and menu selection

SYNOPSIS

```
declare initial_mode      fixed binary(31);
declare key               fixed binary(31);
key = xsm_input(initial_mode);
```

DESCRIPTION

This routine is only used if you are writing your own executive. Use `xsm_input` to open the keyboard for either data entry or menu selection.

You specify which mode you wish to be in with the argument `initial_mode`. Possible choices are defined in `smdefs.incl.pl1`. They are:

- IN_AUTO JAM checks whether you specified the screen to begin menu mode or data entry mode (See Author's Guide).
- IN_DATA Start in data entry mode.
- IN_MENU Start in menu mode.

In most cases you will want to use IN_AUTO mode. Use IN_DATA or IN_MENU if you wish to override the setting that you specified via the Screen Editor.

This routine calls `xsm_getkey` to get and interpret keyboard entry. While in data entry mode ASCII data is entered into fields on the screen, subject to any restrictions or edits that were defined for the fields. The routine returns to the calling program when it encounters a logical key, when a "return entry" field is filled or tabbed from, or a key with the return bit set in the routing table.

If the logical value returned by `xsm_getkey` is TRANSMIT, EXIT, HELP, or a cursor position key, the processing is determined by a routing table. The routing options are set with `xsm_keyoption`. See `xsm_keyoption` for more information.

This function replaces version 4.0 `xsm_choice`, `xsm_menu_proc`, and `xsm_openkeybd`. These functions only exist in your version 5.0 library for backward compatibility. We strongly suggest that you do not use them in the future.

RETURNS

The key hit by the end-user that terminated the call to `xsm_input`, or the first character of the selected menu item.

inquire

obtain value of a global integer variable

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare which              fixed binary(31);
declare value              fixed binary(31);
value = xsm_inquire(which);
```

DESCRIPTION

This function is used to obtain the current integer value of a global variable. The desired variable is specified by `which`. If the value of `which` is a true/false (the flag is on or off) value then `xsm_inquire` returns `1` for true and `0` for false. If you wish to modify a global integer value use `xsm_iset`. The permissible values for `which` are defined in `smglobs.incl.pl1`. The following values are available:

Mnemonic	*Meaning*
I_NODISP	In non-display mode? (T/F). Initially FALSE, setting TRUE causes no further changes to the actual display, although JAM's internal screen image is kept up to date. This was release 4's `sm_do_not_display` flag.
I_INSMODE	In insert mode? (T/F).
I_INXFORM	In JAM screen editor? (T/F) Field validation routines are generally still called when in editor; they can check this flag to disable certain features.
I_MXLINES	Number of lines available for use by JAM on the hardware display.
I_MXCOLMS	Number of columns available for use by JAM on the hardware display.
I_NLINES	Maximum number of lines available on the current screen, not including the status line.
I_NCOLMS	Maximum number of columns available on the current screen, not including the status line.
I_INHELP	Help screen is currently displayed? (T/F)

Mnemonic	*Meaning*
I_BSNESS	Screen manager is in control of display? (T/F). (Replaces rel. 4 inbusiness function).
I_BLKFLGS	Block mode is turned on? (T/F)
SC_VFLINE	First screen line of viewport (0–based).
SC_VFCOLM	First screen column of viewport (0–based).
SC_VNLINE	Number of lines in viewport.
SC_VNCOLM	Number of columns in viewport.
SC_VOLINE	Line offset of viewport.
SC_VOCOLM	Column offset of viewport.
SC_NLINE	Number of lines in screen.
SC_NCOLM	Number of columns in screen.
SC_CLINE	Current line number in screen.
SC_CCOLM	Current column number in screen.
SC_NFLDS	Number of fields on screen.
SC_NGRPS	Number of groups on screen.
SC_BKATTR	Background attributes of screen.
SC_BDCHAR	Border character of screen.
SC_BDATTR	Border attributes of screen.

RETURNS

If the argument corresponds to an integer global variable, the current value of that variable is returned.

1 true, flag is set to on.
0 false, flag is set to off.
–1 otherwise.

RELATED FUNCTIONS

```
value = xsm_finquire(field_number, which);
value = xsm_gp_inquire(group_name, which);
value = xsm_iset(which, newval);
buffer = xsm_pinquire(which);
buffer = xsm_pset(which, newval);
```

intval

get the integer value of a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare value              fixed binary(31);
value = xsm_intval(field_number);
```

DESCRIPTION

This function returns the integer value of the data contained in the field specified by `field_number`. Any punctuation characters in the field, except a leading plus or minus sign, are ignored.

RETURNS

The integer value of the specified field.
0 if the field is not found.

VARIANTS

```
value = xsm_e_intval(field_name, element);
value = xsm_i_intval(field_name, occurrence);
value = xsm_n_intval(field_name);
value = xsm_o_intval(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_itofield(field_number, value);
```

ioccur

insert blank occurrences into an array

SYNOPSIS

```
declare field_number       fixed binary(31);
declare occurrence         fixed binary(31);
declare count              fixed binary(31);
declare lines_inserted     fixed binary(31);
lines_inserted = xsm_o_ioccur(field_number, occurrence, count);
```

DESCRIPTION

NOTE: This function only exists in the `i_` and `o_` variations. There is no `xsm_ioccur`, since this function applies only to arrays.

Inserts `count` blank occurrences before the specified occurrence, moving that occurrence and all following occurrences down. If inserting that many would move an occurrence past the end of its array, fewer will be inserted. If the array is scrollable, then this function may allocate up to `count` new occurrences. This function never increases the maximum number of occurrences an array can contain; `xsm_sc_max` does that. If `count` is negative, occurrences will be deleted instead, subject to limitations described in the page for `xsm_doccur`. In addition, this function never inserts more blank occurrences than the number of blank occurrences following the last non–blank occurrence (that is, it won't push data off the end of an array).

If `occurrence` is zero, the occurrence used is that of `field_number`. If `occurrence` is nonzero, however, it is taken relative to the first field of the array in which `field_number` occurs.

Any clearing–unprotected synchronized arrays will have the same operations performed on them as the referenced array. Synchronized arrays that are protected from clearing will remain constant. Therefore, a protected array may be used to number a list of data stored in a non–protected synchronized array as it grows and shrinks.

This function is normally bound to the INSERT LINE key.

RETURNS

–1 if the field or occurrence number is out of range.
–3 if insufficient memory is available.
otherwise, the number of occurrences actually inserted (zero or more).

VARIANTS

```
lines_inserted = xsm_i_ioccur(field_name, occurrence, count);
```

is_no

test field for no

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_is_no(field_number);
```

DESCRIPTION

The first character of the field contents specified by `field_number` is compared with the first letter of the `SM_NO` entry in the message file, ignoring case. If they match this function will return a 1 for true. If they do not match for any reason, the function returns a 0 for failure. There is no way to tell if the failure is due to a `Y` in the field or because of some other problem. If you wish to check for a `Y` response use `xsm_is_yes`.

This function is ordinarily used with one-letter fields possessing the yes/no character edit. In this case, the only characters allowed in the field are `y`, `n`, or space (which means `n`). Unlike other functions, `xsm_is_no` does *not ignore leading blanks*.

RETURNS

1 if the field's first character matches the first character of the `SM_NO` entry in the message file.
0 otherwise.

VARIANTS

```
status = xsm_e_is_no(field_name, element);
status = xsm_i_is_no(field_name, occurrence);
status = xsm_n_is_no(field_name);
status = xsm_o_is_no(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_is_yes(field_number);
```

is_yes

test field for yes

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_is_yes(field_number);
```

DESCRIPTION

The first character of the field contents specified by `field_number` is compared with the first letter of the `SM_YES` entry in the message file, ignoring case. If they match this function will return a 1 for true. If they do not match for any reason, the function returns a 0 for failure. There is no way to tell if the failure is due to an `N` in the field or because of some other problem. If you wish to check for an `N` response use `xsm_is_no`.

This function is ordinarily used with one–letter fields possessing the yes/no character edit. In this case, the only characters allowed in the field are `y`, `n`, or space (which means `n`). Unlike other functions, `xsm_is_yes` does *not ignore leading blanks*.

RETURNS

1 if the field's first character matches the first character of the `SM_YES` entry in the message file.
0 otherwise.

VARIANTS

```
status = xsm_e_is_yes(field_name, element);
status = xsm_i_is_yes(field_name, occurrence);
status = xsm_n_is_yes(field_name);
status = xsm_o_is_yes(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_is_no(field_number);
```

isabort

test and set the abort control flag

SYNOPSIS

```
declare flag                fixed binary(31);
declare old_flag            fixed binary(31);
old_flag = xsm_isabort(flag);
```

DESCRIPTION

Use `xsm_isabort` to set the abort flag to the value of `flag`, and return the old value. `flag` must be one of the following as defined in `smdefs.incl.pl1`:

Flag	*Meaning*
`ABT_ON`	set abort flag
`ABT_OFF`	clear abort flag
`ABT_DISABLE`	turn abort reporting off
`ABT_NOCHANGE`	do not alter the flag

Abort reporting is intended to provide a quick way out of processing in the JAM library, which may involve nested calls to `xsm_input`. The triggering event is the detection of an abort condition by `xsm_getkey`, either an ABORT keystroke or a call to this function with `ABT_ON` (such as from an asynchronous function).

This function enables application code to verify the existence of an abort condition by testing the flag, as well as to establish one.

RETURNS

The previous value of the abort flag.

iset

change value of integer global variable

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare which              fixed binary(31);
declare newval             fixed binary(31);
declare value              fixed binary(31);
value = xsm_iset(which, newval);
```

DESCRIPTION

JAM has a number of global parameters and settings. This function is used to modify the current value of integer globals. The variable to change is specified by `which`. The new value is specified by `newval`. If you wish to get the value of a global integer use `xsm_inquire`.

The permissible values for the argument `which` are defined in the header file `smglobs.incl.pl1`. The following values are available:

Mnemonic	*Quantity*	*Meaning*
I_NODISP	0	Disable updating of display.
	1	Enable updating of display.
I_INSMODE	0	Enter overtype mode.
	1	Enter insert mode.

RETURNS

If `which` is one of the permissible values, the former value of the appropriate variable is returned.
1 True, the flag was set to on.
0 False, the flag was set to off.
-1 otherwise.

RELATED FUNCTIONS

```
value = xsm_finquire(field_number, which);
```

```
value = xsm_gp_inquire(group_name, which);
value = xsm_inquire(which);
buffer = xsm_pinquire(which);
buffer = xsm_pset(which, newval);
```

isselected

determine whether a radio button or checklist occurrence has been selected

SYNOPSIS

```
declare group_name         char(256) varying;
declare group_occurrence   fixed binary(31);
declare status             fixed binary(31);
status = xsm_isselected(group_name, group_occurrence);
```

DESCRIPTION

This function lets you check to see whether or not a specific occurrence within a check list or radio button has been selected. The selection is referenced by the group name and occurrence number. If the occurrence is selected, `xsm_isselected` returns a `1`. A 0 is returned if the occurrence is not selected. See the Author's Guide for a more detailed discussion of groups.

Radio button and checklist occurrences are selected by using `xsm_select`. Using `xsm_select` on a radio button occurrence causes the current selection to be deselected. Checklist occurrences are deselected with `xsm_deselect`.

RETURNS

-1 arguments do not reference a checklist or radio button occurrence.
0 not selected.
1 selected.

RELATED FUNCTIONS

```
status = xsm_deselect(group_name, group_occurrence);
length = xsm_getfield(buffer, field_number);
value = xsm_intval(field_number);
status = xsm_select(group_name, group_occurrence);
```

issv

determine if a screen is in the saved list

SYNOPSIS

```
declare screen_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_issv(screen_name);
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. This function searches the save list for a single screen and returns 1 is the screen is found (See `xsm_svscreen`).

This function is generally called by applications at screen entry to avoid re-acquiring data (via a database query) for previously saved screens. To accomplish this, first use `xsm_svscreen` to add the screen to the save list upon screen exit. Next, use `xsm_issv` to check the save list upon screen entry. If the screen is on the save list, you know that it has been previously displayed.

RETURNS

1 if the screen is in the saved list.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_svscreen(screen_list, count);
```

itofield

write an integer value to a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare value              fixed binary(31);
declare status             fixed binary(31);
status = xsm_itofield(field_number, value);
```

DESCRIPTION

The integer passed to `xsm_itofield` is converted to characters and placed in the specified field. A number longer than the field will be truncated, on the left or right, according to the field's justification, without warning.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
status = xsm_e_itofield(field_name, element, value);
status = xsm_i_itofield(field_name, occurrence, value);
status = xsm_n_itofield(field_name, value);
status = xsm_o_itofield(field_number, occurrence, value);
```

RELATED FUNCTIONS

```
value = xsm_intval(field_number);
```

jclose

close current window or form under **JAM** Executive control

SYNOPSIS

```
declare status              fixed binary(31);
status = xsm_jclose();
```

DESCRIPTION

The active screen is closed, and the display is restored to the state before the screen was opened. `xsm_jclose` should only be used when the JAM Executive is in use.

In the case of closing a form, `xsm_jclose` pops the form stack and calls `xsm_jform` to display the screen on the top of the form stack.

In the case of closing a window, `xsm_jclose` calls `xsm_close_window`. Since windows are stacked, the effect of closing a window is to return to the previous window. The cursor reappears at the same position it had before the window was opened.

RETURNS

-1 if there is no window open, i.e. if the currently displayed screen is a form (or if there is no screen displayed).
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_close_window();
status = xsm_jform(screen_name);
status = xsm_jwindow(screen_name);
```

jform

display a screen as a form under **JAM** control

SYNOPSIS

```
declare screen_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_jform(screen_name);
```

DESCRIPTION

This function must be used with the JAM Executive. If you are not using the JAM Executive, use `xsm_r_form` or one of its variants. If you wish to display a window under JAM control, use `xsm_jwindow`.

This function displays the named screen as a form. You may close the form with `xsm_jclose`, or leave the task to the JAM Executive (e.g., when the user presses the EXIT key). Bringing up a screen as a form causes the previously displayed form and windows to be discarded, and their memory freed. The new form is placed on top of the JAM's form stack.

The difference between `xsm_jform` and `xsm_r_form`, other than the function arguments, is that only `xsm_jform` manipulates the form stack. Since `xsm_jform` calls `xsm_r_form`, refer to `xsm_r_form` for information on other details, such as how the screen to be displayed is found.

The character string `screen_name` uses the same format as that of a JAM control string that displays a form. In addition to the screen's name, you may optionally specify the position of the form on the physical display, the size of the viewport, and which portion of the form will be positioned in the viewport's top–left corner. See the Authoring Reference in the Author's Guide for details of viewport positioning. The following are all legal strings:

```
status = xsm_jform('form');
```

Display form's first row and column at the top–left corner of the physical display.

```
status = xsm_jform('(20,10)form');
```

Display form's first row and column at the 20th row and 10th column of the physical display.

```
status = xsm_jform('(20,10,15,8)form');
```

Display the first row and column of the form at the 20th row and 10th column of the physical display in viewport that is 15 rows by 8 columns.

A form may be larger than the viewport. If the viewport does not fit on the screen where indicated, JAM will attempt to place it entirely on the display at a different location. If

you specify a viewport that is larger than the physical display, the viewport will be the size of the physical display. If you wish to change the viewport size after the window is displayed, use `xsm_viewport`.

RETURNS

0 if no error occurred.
–1 if the screen file's format is incorrect.
–2 if the screen cannot be found.
–4 if, after the display has been cleared, the screen cannot be successfully displayed because of a read error.
–5 if, after the display was cleared, the system ran out of memory.

RELATED FUNCTIONS

```
status = xsm_r_form(screen_name);
status = xsm_jwindow(screen_name);
```

jplcall

execute a JPL jpl procedure

SYNOPSIS

```
declare jplcall_text       char(256) varying;
declare return_value       fixed binary(31);
return_value = xsm_jplcall(jplcall_text);
```

DESCRIPTION

This function executes a JPL procedure precisely as if the following JPL statement were executed from within a JPL procedure:

```
jpl jplcall_text
```

For example, if the value of `jplcall_text` were:

```
verifysal :name 50000
```

then

and

```
jpl verifysal :name 50000
```

would be equivalent. See the JPL Programmer's Guide for further information on the JPL `jpl` command.

RETURNS

-1 if the procedure could not be loaded.
Otherwise, the value returned by the JPL procedure.

jplload

execute the JPL load command

SYNOPSIS

```
declare module_name_list   char(256) varying;
declare status             fixed binary(31);
status = xsm_jplload(module_name_list);
```

DESCRIPTION

This function is the PL/1 interface to the JPL `load` command. Use this command to load one or more modules into memory.

The character string `module_name_list` may be one or more module names. Separate module names with a space.

Calling `xsm_jplload` has precisely the same effect as using the JPL `load` command. See the JPL Programmer's Guide for further information on the JPL `load` command.

Use `xsm_jplunload` to remove a module from memory.

RETURNS

–1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_jplpublic(module_name_list);
status = xsm_jplunload(module_name);
```

jplpublic

execute the JPL public command

SYNOPSIS

```
declare module_name_list   char(256) varying;
declare status             fixed binary(31);
status = xsm_jplpublic(module_name_list);
```

DESCRIPTION

This function is the PL/1 interface to the JPL `public` command. Use this command to load one or more modules into memory.

The character string `module_name` may be one or more module names. Separate module names with a space.

Calling `xsm_jplpublic` has precisely the same effect as using the JPL `public` command. See the JPL Programmer's Guide for further information on the JPL `public` command.

Use `xsm_jplunload` to remove a module from memory.

RETURNS

-1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_jplload(module_name_list);
status = xsm_jplunload(module_name);
```

jplunload

execute the JPL unload command

SYNOPSIS

```
declare module_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_jplunload(module_name);
```

DESCRIPTION

This function is the PL/1 interface to the JPL `unload` command. Use this command to remove one or more modules from memory. Modules are read into memory by using either `xsm_jplpublic` or `xsm_jplload` or via the corresponding JPL commands.

Calling `xsm_jplunload` has precisely the same effect as using the JPL `unload` command. See the JPL Programmer's Guide for further information on the JPL `unload` command.

The character string `module_name` may be one or more module names. Separate module names with a space.

RETURNS

–1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_jplload(module_name_list);
status = xsm_jplpublic(module_name_list);
```

jtop
start the **JAM** Executive

SYNOPSIS

```
declare screen_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_jtop(screen_name);
```

DESCRIPTION

All applications using the JAM Executive must include a call to `xsm_jtop`. This function starts the JAM Executive. The argument `screen_name` is the name of the first screen that your application displays. It will be displayed as a form. Once `xsm_jtop` is called the JAM Executive is in control until the user exits the application.

The JAM Executive makes calls to various JAM functions that handle all of the tasks needed to control the flow of an application such as opening the keyboard for input, opening and closing forms and windows, and processing all control strings.

If you do not use `xsm_jtop` you will have to write your own procedures to control the flow of your application. See the JAM Development Overview for a more detailed discussion of the JAM Executive.

RETURNS

0 Always.

jwindow

display a window at a given position under **JAM** control

SYNOPSIS

```
declare screen_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_jwindow(screen_name);
```

DESCRIPTION

This function must be used with the JAM Executive. If you are not using the JAM Executive, use `xsm_r_window` or one of its variants. If you wish to display a form under JAM control, use `xsm_jform`.

This function displays the named screen as a window, by calling `xsm_r_window`. You may close the window with a call to `xsm_jclose`, or leave the task to the JAM Executive (e.g., when the user presses the EXIT key).

There is currently no difference between `xsm_jwindow` and `xsm_r_window` except for their arguments (although `xsm_jwindow` is not supported unless the JAM Executive is in use). See the description of `xsm_r_window` for the details of the behavior of `xsm_jwindow`.

The character string `screen_name` uses a format similar to that of a JAM control string that displays a window. Use a single ampersand to specify a stacked window and a double ampersand to specify a sibling window. If the ampersand is omitted, then the screen will be opened as a stacked window. In addition to the screen's name, you may optionally specify the position of the window on the physical display, the size of the viewport, as well as which portion of the window will be positioned in the viewport's top–left corner. The positioning and sizing syntax is identical to that of `xsm_jform`. See `xsm_jform` for examples of acceptable strings.

RETURNS

0 if no error occurred during display of the screen
–1 if the screen file's format is incorrect
–2 if the form cannot be found
–3 if the system ran out of memory but the previous screen was restored

RELATED FUNCTIONS

```
status = xsm_jclose();
status = xsm_jform(screen_name);
```

```
status = xsm_r_window(screen_name, start_line, start_column);
```

keyfilter

control keystroke record/playback filtering

SYNOPSIS

```
declare flag                fixed binary(31);
declare old_flag            fixed binary(31);
old_flag = xsm_keyfilter(flag);
```

DESCRIPTION

This function turns the keystroke record/playback mechanism of `xsm_getkey` on (`flag` = 1) or off (`flag` = 0). If no key recording or playback function has been installed, turning the mechanism on has no effect.

It returns a flag indicating whether recording was previously on or off.

RETURNS

The previous value of the filter flag.

RELATED FUNCTIONS

```
key = xsm_getkey();
```

keyhit

test whether a key has been typed ahead

SYNOPSIS

```
declare interval          fixed binary(31);
declare status            fixed binary(31);
status = xsm_keyhit(interval);
```

DESCRIPTION

This function checks whether a key has already been hit; if so, it returns 1 immediately. If not, it waits for the indicated interval and checks again. The key (if any is struck) is *not* read in, and is available to the usual keyboard input routines.

`interval` is in tenths of seconds; the exact length of the wait depends on the granularity of the system clock, and is hardware– and operating–system dependent. JAM uses this function to decide when to call the user–supplied asynchronous function.

If the operating system does not support reads with timeout, this function ignores the interval and only returns 1 if a key has been typed ahead.

RETURNS

0 if no key is available,
non–0 otherwise.

RELATED FUNCTIONS

```
key = xsm_getkey();
```

keyinit

initialize key translation table

SYNOPSIS

```
declare key_address        bit(0);
declare status             fixed binary(31);
status = xsm_keyinit(key_address);
```

DESCRIPTION

This routine is called by `xsm_initcrt` as part of the initialization process, but it can also be called by an application program (either before or after `xsm_initcrt`) to install a memory-resident key translation file.

To install a memory-resident key translation file, `key_address` must contain the address of a key translation table created using the `key2bin` and `bin2pl1` utilities.

RETURNS

0 if the key file is successfully installed.
Program exit if the key file is invalid.

VARIANTS

```
status = xsm_n_keyinit(key_file);
```

keylabel

get the printable name of a logical key

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare buffer              char(256) varying;
declare key                 fixed binary(31);
buffer = xsm_keylabel(key);
```

DESCRIPTION

Returns the label defined for `key` in the key translation file; the label is usually what is printed on the key on the physical keyboard. If there is no such label, returns the name of the logical key from the following table. Here is a list of key mnemonics:

Logical Key Mnemonics							
EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL	BACK
HOME	DELE	INS	LP	FERA	CLR	SPGU	SPGD
LSHF	RSHF	LARR	RARR	DARR	UARR	REFR	EMOH
INSL	DELL	ZOOM	SFTS	MTGL	VWPT	MOUS	
PF1-PF24		SPF1-SPF24		APP1-APP24		SFT1-SFT24	

If the key code is invalid (not one defined in `smkeys.incl.pl1`), this function returns an empty string.

RETURNS

A string naming the key, or an empty string if it has no name.

keyoption

set cursor control key options

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare key                 fixed binary(31);
declare mode                fixed binary(31);
declare newval              fixed binary(31);
declare oldval              fixed binary(31);
oldval = xsm_keyoption(key, mode, newval);
```

DESCRIPTION

Use `xsm_keyoption` to alter at run-time the behavior of `xsm_input` when a particular key is pressed. The default values for key options are built in to JAM. This function only works with cursor control keys. Cursor control keys include all JAM logical keys, *except* for PF, SPF, and APP keys. See "Key File" in the Configuration Guide.

There are three different possible values for `mode`: `KEY_ROUTING`, `KEY_GROUP`, and `KEY_XLATE`. The mnemonics that they use are defined in `smkeys.incl.pl1`. All of these modes draw on the following values for `key`.

Logical Key Mnemonics							
EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL	BACK
HOME	DELE	INS	LP	FERA	CLR	SPGU	SPGD
LSHF	RSHF	LARR	RARR	DARR	UARR	REFR	EMOH
INSL	DELL	ZOOM	SFTS	MTGL	VWPT	MOUS	

■`KEY_ROUTING`

Allows access to the EXECUTE and RETURN bits of the routing table. This mode is generally used to disable a key or to control explictly what action is taken when a key is hit. The following mnemonics may be assigned to `newval`:

1. `KEY_IGNORE` Disables `key`. JAM does nothing when `key` is struck.
2. `EXECUTE` The action normally associated with `key` is executed. May be ored with `RETURN`.

3. RETURN No action is performed, but the function returns to the caller in your code. Used to gain direct control of key's action. May be ored with EXECUTE.

■KEY_GROUP

Allows access to the group action bits. Use this function to control the action of the cursor when it is within a group. The following values may be assigned to newval:

1. VF_GROUP Obey group semantics. Hitting key will cause the cursor to move to the next field within the group in the indicated direction. If this mnemonic is ored with VF_CHANGE the cursor will exit the group in the indicated direction.
2. VF_CHANGE This value has no effect, unless it is ored with VF_GROUP. In this case the cursor will exit the group in the indicated direction.
3. 0 Assigning zero to newval will cause key to treat a field within a group as if it were not part of a group.
4. VF_OFFSCREEN Offscreen data will scroll onscreen from the direction indicated.
5. VF_NOPROT key will move cursor into a field protected from tabbing.

■KEY_XLATE

Allows access to the cursor table. Use this routine to assign key the action preformed by newval. newval may be any of the logical keys listed in the table above. This can often replace a user–supplied key change function.

RETURNS

–1 if some parameter is out of range.
the old value otherwise.

keyset
open a keyset

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

declare name                char(256) varying;
declare scope               fixed binary(31);
declare status              fixed binary(31);
status = xsm_r_keyset(name, scope);


declare address             bit(0);
declare scope               fixed binary(31);
declare status              fixed binary(31);
status = xsm_d_keyset(address, scope);
```

DESCRIPTION

Use `xsm_d_keyset` and `xsm_r_keyset` to display a keyset. The parameter `name` is the name of the keyset. `scope` must be one of the mnemonics listed in `smsoftk.incl.pl1`. Application programs will normally use scope `KS_APPLIC`. Values for `scope` are defined in `smsoftk.incl.pl1`. For a more detailed explanation of scope see the Key Set chapter of the Author's Guide.

If there is currently a keyset of the specified scope the name of that keyset is compared with the name passed. If they are the same the present routine returns immediately. This means that if you want to "refresh" a keyset with a new copy from disk, you must first close the keyset with a call to `xsm_c_keyset`.

If the call is not successful then the current keyset remains displayed and an error message is posted to the end–user, except where noted otherwise.

The most commonly used variant is `xsm_r_keyset`. You do not need to know where the keyset resides because `xsm_r_keyset` searches for you. It looks first in the memory resident form list, next in any open libraries, then on disk in the directory specified by the argument to `xsm_initcrt`, and finally in the directories specified by `SMPATH`. Keyset files may be mixed freely with screen files in the screen list and in libraries.

You may save processing time by using `xsm_d_keyset` to display a memory–resident keyset. `address` is a pointer to the keyset in memory. Use the utility `bin2pl1` to create

program data structures, from disk–based keysets, that you can compile into your application.

To close a keyset use `xsm_c_keyset`.

RETURNS

0 If no error occurred during display of the keyset.
–1 If the format incorrect (not a keyset).
–2 if the keyset cannot be found. No message is posted to the end–user.
–3 If the terminal doesn't support soft keys (or scope out of range).
–4 If there is a read error.
–5 If there is a malloc failure.

kscscope

query current keyset scope

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

declare scope              fixed binary(31);
scope = xsm_kscscope();
```

DESCRIPTION

This routine returns the scope of the current keyset or `-1` if no keyset is currently active.

This function can be used to determine whether or not the application keyset (as opposed to the system keyset) is currently displayed.

Values for `scope` are defined in `smsoftk.incl.pl1`.

RETURNS

Current scope, or
-1 if not found.

RELATED FUNCTIONS

```
status = xsm_ksinq(scope, number_keys, number_rows,
         current_row, maximum_len, keyset_name);
status = xsm_skvinq(scope, value, occurrence, attribute,
         label1, label2);
```

ksinq

inquire about keyset information

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

declare scope              fixed binary(31);
declare number_keys        fixed binary(31);
declare number_rows        fixed binary(31);
declare current_row        fixed binary(31);
declare maximum_len        fixed binary(31);
declare keyset_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_ksinq(scope, number_keys, number_rows,
         current_row, maximum_len, keyset_name);
```

DESCRIPTION

Use this routine to obtain the name, number of rows, number of items within a row, and current row of a keyset currently in memory. You supply the keyset's `scope` and five addresses to hold the information returned by `xsm_skinq`. `scope` must be one of the mnemonics defined in `smsoftk.incl.pl1`.

The function places the number of rows in the keyset in `number_row`, the number of soft keys per row in `number_keys`, and the current row number in `current_row`. The name of the keyset is placed in the pre-allocated buffer `keyset_name`. The size of `keyset_name` is specified by `maximum_len`. If the name of the keyset in longer then `keyset_name`, then `xsm_ksinq` fills the buffer to the end without adding a null character, otherwise a null character is added to the end of the string. The null pointer may be used for any or all of the parameters about which you do not desire information.

RETURNS

0 if information is returned.
-1 if there is no active keyset for the given scope.
-2 for an invalid scope.

RELATED FUNCTIONS

```
scope = xsm_kscscope();
size = xsm_skinq(scope, row, softkey, value, display_attribute,
       label1, label2);
```

```
status = xsm_skvinq(scope, value, occurrence, attribute,
         label1, label2);
```

ksoff

turn off soft key labels

SYNOPSIS

```
call xsm_ksoff();
```

DESCRIPTION

When a keyset is opened with any of the library routines, the labels are automatically displayed. If you do not wish to display the labels at any point within your application, use `xsm_ksoff` to turn the display off.

If you wish to turn them the label display back on, use `xsm_kson`.

RELATED FUNCTIONS

```
call xsm_kson();
```

kson

turn on soft key labels

SYNOPSIS

```
call xsm_kson();
```

DESCRIPTION

Normally, keyset labels are displayed when a keyset is called up. The only way the display can be turned off is with the library routine, `xsm_ksoff`. Use this routine to turn the label display back on.

RELATED FUNCTIONS

```
call xsm_ksoff();
```

l_close
close a library

SYNOPSIS

```
declare lib_desc          fixed binary(31);
declare status            fixed binary(31);
status = xsm_l_close(lib_desc);
```

DESCRIPTION

Closes the library indicated by `lib_desc` and frees all associated memory. The library descriptor is a number returned by a previous call to `xsm_l_open`.

RETURNS

-1 is returned if the library file could not be closed.
-2 is returned if the library was not open.
0 is returned if the library was closed successfully.

RELATED FUNCTIONS

```
status = xsm_l_at_cur(lib_desc, screen_name);
status = xsm_l_form(lib_desc, screen_name);
lib_desc = xsm_l_open(lib_name);
status = xsm_l_window(lib_desc, screen_name, start_line,
         start_column);
```

l_open
open a library

SYNOPSIS

```
declare lib_name          char(256) varying;
declare lib_desc          fixed binary(31);
lib_desc = xsm_l_open(lib_name);
```

DESCRIPTION

You must use `xsm_l_open` to open a library before you use a JPL module, a keyset, or a screen that is stored in the library. Use the utility `formlib` to create a library. (See the JAM Utilites Guide).

This routine allocates space in which to store information about the library, leaves the library file open, and returns a descriptor identifying the library. The descriptor may subsequently be used by `xsm_l_window` and related functions, to display screens stored in the library. The library can also be referenced implicitly by `xsm_r_window`, `xsm_r_keyset`, and `xsm_jplcall`, as well as related functions, which search all open libraries.

The library file is sought in all the directories identified by `SMPATH` and the parameter to `xsm_initcrt`. If you define the `SMFLIBS` variable in your setup file as a list of library names `xsm_l_open` will automatically be called for those libraries. The `xsm_r_` routines will then search in the specified libraries.

Several libraries may be kept open at once. This may cause problems on systems with severe limits on memory or simultaneously open files.

RETURNS

–1 if the library cannot be opened or read.
–2 if too many libraries are already open.
–3 if the named file is not a library.
–4 if insufficient memory is available.
Otherwise, a non–negative integer that identifies the library file.

RELATED FUNCTIONS

```
return_value = xsm_jplcall(jplcall_text);
status = xsm_jplload(module_name_list);
status = xsm_jplpublic(module_name_list);
status = xsm_l_at_cur(lib_desc, screen_name);
```

```
status = xsm_l_close(lib_desc);
status = xsm_l_form(lib_desc, screen_name);
status = xsm_l_window(lib_desc, screen_name, start_line,
         start_column);
status = xsm_r_at_cur(screen_name);
status = xsm_r_form(screen_name);
status = xsm_r_keyset(name, scope);
status = xsm_r_window(screen_name, start_line, start_column);
```

last

position the cursor in the last field

SYNOPSIS

```
call xsm_last();
```

DESCRIPTION

Use this function to place the cursor at the first enterable position of the last tab–unprotected field of the current screen. If the last field unprotected from tabbing is right justified, the cursor is placed in the rightmost position of the field. By the same token, if the last unprotected field is left justified, the cursor is placed in the leftmost position of the field.

Unlike `xsm_home`, `xsm_last` will not reposition the cursor if the screen has no unprotected fields.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

This function is called when the JAM logical key EMOH is struck.

RELATED FUNCTIONS

```
call xsm_backtab();
field_number = xsm_home();
call xsm_nl();
call xsm_tab();
```

lclear

erase LDB entries of one scope

SYNOPSIS

```
declare scope              fixed binary(31);
declare status             fixed binary(31);
status = xsm_lclear(scope);
```

DESCRIPTION

This function erases the values stored in the local data block for all names having a scope of the argument `scope`. Legal values for scope are between `1` and `9`. Constant variables having scope `1` *can* be erased.

Refer to the LDB chapter of the Programmer's Guide for a discussion of the scope of LDB entries.

RETURNS

-1 if scope is invalid.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_lreset(file_name, scope);
```

ldb_init

initialize (or reinitialize) the local data block

SYNOPSIS

```
call xsm_ldb_init();
```

DESCRIPTION

This function creates an empty index of named data items by reading the data dictionary, then loads values into them from initialization files. Data Dictionary entries with a scope of 0 are not loaded into the LDB. There is no LDB prior to the first execution of this function.

Selected parts of the LDB, namely those assigned a certain scope, can be reinitialized using `xsm_lclear` or `xsm_lreset`.

This function is called explicitly in `jmain.pl1` and `jxmain.pl1`. Other functions that affect its behavior, such as `xsm_dicname` and `xsm_ininames`, should be called first.

RELATED FUNCTIONS

```
status = xsm_dicname(dic_name);
status = xsm_ininames(name_list);
status = xsm_lreset(file_name, scope);
```

leave

prepare to leave a **JAM** application temporarily

SYNOPSIS

```
call xsm_leave();
```

DESCRIPTION

At times it may be necessary to leave a JAM application temporarily. For example you may need to escape to the command interpreter or to execute some graphics functions. In such a case, the terminal and its operating system channel need to be restored to their normal states.

This function should be called before leaving. It clears the physical screen (but not the internal screen image); resets the operating system channel; and resets the terminal (using the RESET sequence found in the video file).

RELATED FUNCTIONS

```
call xsm_return();
```

length

get the maximum length of a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare field_length       fixed binary(31);
field_length = xsm_length(field_number);
```

DESCRIPTION

This function returns the maximum length of the field specified by `field_number`. If the field is shiftable, its maximum shifting length is returned. This length is as defined in the JAM Screen Editor, and has no relation to the current contents of the field. Use `xsm_dlength` to get the length of the contents.

RETURNS

Length of the field.
0 if the field is not found.

VARIANTS

```
field_length = xsm_n_length(field_name);
```

RELATED FUNCTIONS

```
data_length = xsm_dlength(field_number);
```

lngval

get the long integer value of a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare value              fixed binary(31);
value = xsm_lngval(field_number);
```

DESCRIPTION

This function returns the contents of `field_number`, converted to a long integer. All non-digit characters are ignored, except for a leading plus or minus sign.

RETURNS

The long value of the field.
0 if the field is not found.

VARIANTS

```
value = xsm_e_lngval(field_name, element);
value = xsm_i_lngval(field_name, occurrence);
value = xsm_n_lngval(field_name);
value = xsm_o_lngval(field_number, occurrence);
```

RELATED FUNCTIONS

```
value = xsm_intval(field_number);
status = xsm_ltofield(field_number, value);
```

lreset

reinitialize LDB entries of one scope

SYNOPSIS

```
declare file_name          char(256) varying;
declare scope              fixed binary(31);
declare status             fixed binary(31);
status = xsm_lreset(file_name, scope);
```

DESCRIPTION

This function sets local data block entries to values read from `file_name`. The `scope` must be between 1 and 9. References in the file to LDB entries not belonging to `scope` are ignored. All variables belonging to `scope` are cleared before reinitializing. This means that `xsm_lreset` erases variables that are not in the file.

The file may be in the current directory, or in any of the directories listed in the `SMPATH` environment variable. It contains pairs of names with values, each enclosed in quotes. While all whites space outside the quotes is ignored, we recommend for readability that the file have one name–value pair per line. If an entry has multiple occurrences, it may be subscripted in the file. Here are a few sample pairs:

```
"husband"   "Ronald Reagan"
"wife[1]"   "Jane Wyman"
"wife[2]"   "Nancy Davis"
```

If you plan to use this function, we recommend that you group your variables in separate files by scope. You can use `xsm_ininames` to list a number of files for initialization.

RETURNS

–1 if file not found or scope out of range.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_lclear(scope);
```

lstore

copy everything from screen to LDB

SYNOPSIS

```
declare status              fixed binary(31);
status = xsm_lstore();
```

DESCRIPTION

This function copies data from the screen to local data block entries with matching names.

The JAM Executive automatically calls `xsm_lstore` when bringing up a new screen or before closing a window. This function need not be called by application code except under special circumstances.

RETURNS

-3 if sufficient memory is not available.
0 otherwise.

RELATED FUNCTIONS

```
call xsm_allget(respect_flag);
```

ltofield

place a long integer in a field

SYNOPSIS

```
declare field_number        fixed binary(31);
declare value               fixed binary(31);
declare status              fixed binary(31);
status = xsm_ltofield(field_number, value);
```

DESCRIPTION

The long integer passed to this routine is converted to human–readable form and placed in `field_number`. If the number is longer than the field, it is truncated without warning, on the right or left depending on the field's justification.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
status = xsm_e_ltofield(field_name, element, value);
status = xsm_i_ltofield(field_name, occurrence, value);
status = xsm_n_ltofield(field_name, value);
status = xsm_o_ltofield(field_number, occurrence, value);
```

RELATED FUNCTIONS

```
status = xsm_itofield(field_number, value);
value = xsm_lngval(field_number);
```

m_flush

flush the message line

SYNOPSIS

```
call xsm_m_flush();
```

DESCRIPTION

This function forces updates to the message line to be written to the display. This is useful if you want to display the status of an operation with `xsm_d_msg_line`, without flushing the entire display as `xsm_flush` does.

RELATED FUNCTIONS

```
call xsm_flush();
```

max_occur

get the maximum number of occurrences

SYNOPSIS

```
declare field_number       fixed binary(31);
declare maximum            fixed binary(31);
maximum = xsm_max_occur(field_number);
```

DESCRIPTION

This function returns the maximum number of occurrences that the array can hold as defined in the JAM Screen Editor or by `xsm_sc_max`. If you wish to find out the highest occurrence number of an array that actually contains data, use `xsm_num_occurs`.

RETURNS

0 if the field designation is invalid.
1 for a non–scrollable single field.
The number of elements in a non–scrollable array.
The maximum number of occurrences in a scrollable array.

VARIANTS

```
maximum = xsm_n_max_occur(field_name);
```

RELATED FUNCTIONS

```
number = xsm_num_occurs(field_number);
```

mnutogl

switch between menu mode and data entry mode on a dual–purpose screen

SYNOPSIS

```
declare screen-mode        fixed binary(31);
declare old_mode           fixed binary(31);
old_mode = xsm_mnutogl(screen_mode);
```

DESCRIPTION

JAM supports the use of a single screen as both a menu and a data entry screen, but the screen must be in one or the other "mode" at any given moment. This function can be used to change the mode of the screen and to test which mode the screen is in currently. The `mode` argument may have one of four values as defined in `smdefs.incl.pl1`:

Value	*Meaning*
`IN_AUTO`	No action (generally used just to test the return value).
`IN_DATA`	Change the screen to data entry mode.
`IN_MENU`	Change the screen to menu mode.
`IN_TOGL`	Toggle the screen from one mode to the other (akin to the `MTGL` logical key).

This function is similar to the built–in control function `jm_mnutogl`.

RETURNS

The mode that the screen was in before the function was called (`IN_DATA` or `IN_MENU`.) –1 if the mode specification is invalid.

msg

display a message at a given column on the status line

SYNOPSIS

```
declare column              fixed binary(31);
declare disp_length         fixed binary(31);
declare text                char(256) varying;
call xsm_msg(column, disp_length, text);
```

DESCRIPTION

The message is merged with the current contents of the status line, and displayed beginning at `column`. `disp_length` gives the number of characters to display.

On terminals with onscreen attributes, the column position may need to be adjusted to allow for attributes embedded in the status line. Refer to `xsm_d_msg_line` for an explanation of how to embed attributes and function key names in a status line message.

This function is called by the function that updates the cursor position display (see `xsm_c_vis`).

RELATED FUNCTIONS

```
call xsm_d_msg_line(message, display_attribute);
```

msg_get

find a message given its number

SYNOPSIS

```
%include 'smerror.incl.pl1';

declare buffer              char(256) varying;
buffer = xsm_msg_get(number);
```

DESCRIPTION

The messages used by JAM library routines are stored in binary message files, which are created from text files using the JAM utility, `msg2bin`. Use `xsm_msgread` to load message files for use by this function.

This function takes the `number` of the message desired and returns the message, or a less informative string if the message number cannot be matched.

Messages are divided into classes based on their numbers, with up to 4096 messages per class. The message class is the message number divided by 4096, and the message offset within the class is the message number *modulo* 4096. Predefined JAM message numbers and classes are defined in `smerror.incl.pl1`.

RETURNS

The desired message, if found
otherwise, the message class and number, as a string

RELATED FUNCTIONS

```
buffer = xsm_msgfind(number);
status = xsm_msgread(code, class, mode, arg);
```

msgfind

find a message given its number

SYNOPSIS

```
%include 'smerror.incl.pl1';

declare buffer              char(256) varying;
declare number              fixed binary(31);
buffer = xsm_msgfind(number);
```

DESCRIPTION

This function takes the `number` of a Screen Manager message, and returns the message string. It is identical to `xsm_msg_get`, except that it returns zero if the message number is not found.

Screen Manager message numbers are defined in `smerror.incl.pl1`.

RETURNS

The message
0 if the message number is out of range

RELATED FUNCTIONS

```
buffer = xsm_msg_get(number);
status = xsm_msgread(code, class, mode, arg);
```

msgread

read message file into memory

SYNOPSIS

```
%include 'smerror.incl.pl1';

declare code                char(256) varying;
declare class               fixed binary(31);
declare mode                fixed binary(31);
declare arg                 char(256) varying;
declare status              fixed binary(31);
status = xsm_msgread(code, class, mode, arg);
```

DESCRIPTION

Reads a single set of messages from a binary message file into memory, after which they can be accessed using `xsm_msg_get` and `xsm_msgfind`. The `code` argument selects a single message class from a file that may contain several classes:

Code	*Class*	*Message Type*
SM	SM_MSGS	Screen Manager
FM	FM_MSGS	Screen Editor
JM	JM_MSGS	JAM run-time
JX	JX_MSGS	Data Dictionary & Control Strings
UT	UT_MSG	Utilities
(blank)		Undesignated user

`class` identifies a class of messages. Classes 0–7 are reserved for user messages, and several classes are reserved to JAM; see `smerror.incl.pl1`. As messages with the prefix `code` are read from the file, they are assigned numbers sequentially beginning at 4096 times `class`.

`mode` is a mnemonic composed from the following list. The first five indicate where to get the message file; at least one of these must be supplied. The latter four modify the basic action.

Mnemonic	*Action*
MSG_DELETE	Delete the message class and recover its memory.
MSG_DEFAULT	Use the default file defined by the setup variable SMMSGS.
MSG_FILENAME	Use the file named by `arg`.
MSG_ENVIRON	Use the file named in an environment variable named by `arg`.
MSG_MEMORY	Use a memory-resident file whose address is given by arg.
MSG_NOREPLACE	Modifier: do not overwrite previously installed messages.
MSG_DSK	Modifier: leave file open, do not read into memory
MSG_INIT	Modifier: do not use screen manager error reporting.
MSG_QUIET	Modifier: do not report errors.

You can or MSG_NOREPLACE with any mode except MSG_DELETE, to prevent overwriting messages read previously. Error messages will be displayed on the status line, if the screen has been initialized by `xsm_initcrt`; otherwise, they will go to the standard error output. You can or MSG_INIT with the mode to force error messages to standard error. Combining the mode with MSG_QUIET suppresses error reporting altogether.

If you or MSG_DSK with the mode, the messages are not read into memory. Instead the file is left open, and `xsm_msg_get` and `xsm_msgfind` fetch them from disk when requested. If your message file is large, this can save substantial memory; but you should remember to account for operating system file buffers in your calculations.

`arg` contains the environment variable name for MSG_ENVIRON; the file name for MSG_FILENAME; or the address of the memory-resident file for MSG_MEMORY. It may be passed as zero for other modes.

RETURNS

0 if the operation completed successfully.
1 if the message class was already in memory and the mode included MSG_NOREPLACE.
2 if the mode was MSG_DELETE and the message file was not in memory.
-1 if the mode was MSG_ENVIRON and the environment variable was undefined.
-2 if the mode was MSG_ENVIRON or MSG_FILENAME and the message file could not be read from disk; other negative values if the message file was bad or insufficient memory was available.

RELATED FUNCTIONS

```
buffer = xsm_msg_get(number);
buffer = xsm_msgfind(number);
```

mwindow

display a status message in a window

SYNOPSIS

```
declare text                char(256) varying;
declare line                fixed binary(31);
declare column              fixed binary(31);
declare status              fixed binary(31);
status = xsm_mwindow(text, line, column);
```

DESCRIPTION

This function displays `text` in a pop–up window, whose upper left–hand corner appears at `line` and `column`. The line and column are counted from `0`. If `line` is `1`, the top of the window will be on the second line of the display. The window itself is constructed on the fly by the run–time system. No data entry is possible in it, nor is data entry possible in underlying screens as long as it is displayed.

Due to the delayed write feature in JAM, you should call `xsm_flush` to cause the screen to be updated and the message to be displayed, unless you call `xsm_input` directly after the call to `xsm_mwindow`. `xsm_close_window` may be used to close a window called with `xsm_mwindow`.

All the percent escapes for status messages, except %M and %W, are effective. Refer to `xsm_err_reset` for a list and full description. If either `line` or `column` is negative, the window will be displayed according to the rules given for `xsm_r_at_cur`.

RETURNS

–1 if there was a malloc failure.
1 if the text had to be truncated to fit in a window.
0 otherwise.

RELATED FUNCTIONS

```
call xsm_d_msg_line(message, display_attribute);
```

n_

variants that take a field name only

SYNOPSIS

```
declare field_name        char(256) varying;
call xsm_n_...(field_name, ...);
```

DESCRIPTION

The `n_` functions access a field by means of the field/group name. For a complete description of individual functions, look under the related function without `n_` in its name. For example, `xsm_n_amt_format` is described under `xsm_amt_format`. If the named field/group is not on the screen, these functions will attempt to access a similarly named entry in the local data block.

name

obtain field name given field number

SYNOPSIS

```
declare buffer              char(256) varying;
declare field_number        fixed binary(31);
buffer = xsm_name(field_number);
```

DESCRIPTION

Given a field number, `xsm_name` returns a buffer that contains the field name referenced by `field_number`.

RETURNS

The name of the field referenced, if found.
0 otherwise.

nl

position cursor to the first unprotected field beyond the current line

SYNOPSIS

```
call xsm_nl();
```

DESCRIPTION

This function moves the cursor to the next occurrence of an array, scrolling if necessary. Unlike the down–arrow, it will allocate an empty scrolling occurrence if there are no more below but the maximum has not yet been exceeded.

If the current field is not scrolling, the cursor is positioned to the first unprotected field, if any, following the current *line* of the form. If there are no unprotected fields beyond the current field, the cursor is positioned to the first unprotected field of the screen.

If the screen has no unprotected fields at all, the cursor is positioned to the first column of the line following the current line. If the cursor is on the last line of the form, it goes to the top left–hand corner of the screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

This function is ordinarily bound to the RETURN key.

RELATED FUNCTIONS

```
call xsm_backtab();
field_number = xsm_home();
call xsm_last();
call xsm_tab();
```

novalbit

forcibly invalidate a field

SYNOPSIS

```
declare field_number      fixed binary(31);
declare status            fixed binary(31);
status = xsm_novalbit(field_number);
```

DESCRIPTION

Resets the VALIDED bit of the specified field, so that the field will again be subject to validation when it is next exited, or when the screen is validated as a whole.

JAM sets a field's VALIDED bit automatically when the field passes all its validations. The bit is initially clear, and is cleared whenever the field is altered by keyboard input or by a library function such as `xsm_putfield`.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
status = xsm_e_novalbit(field_name, element);
status = xsm_i_novalbit(field_name, occurrence);
status = xsm_n_novalbit(field_name);
status = xsm_o_novalbit(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_fval(field_number);
status = xsm_s_val();
```

null

test if field is null

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_null(field_number);
```

DESCRIPTION

Use `xsm_null` to test a field to see whether it has both the null edit and contains the null character string that has been assigned to that field. See null edits in the Author's Guide.

RETURNS

1 If the field has the null edit and contains the appropriate null character string.
−1 if the field does not exist.
0 otherwise.

VARIANTS

```
status = xsm_e_null(field_name, element);
status = xsm_i_null(field_name, occurrence);
status = xsm_n_null(field_name);
status = xsm_o_null(field_number, occurrence);
```

num_occurs

find the highest numbered occurrence containing data

SYNOPSIS

```
declare field_number       fixed binary(31);
declare number             fixed binary(31);
number = xsm_num_occurs(field_number);
```

DESCRIPTION

This function returns the highest occurrence number of the array specified by `field_number` that actually contains data. The field number may be that of any field with the array.

Most of the time the highest numbered occurrence containing data will be the same as the number of occurrences actually containing data. However, it is possible to have blank occurrences preceding occurrences containing data.

This count is different from the maximum capacity of an array, which you can retrieve with `xsm_max_occur`.

RETURNS

The highest numbered occurrence containing data.
0 if there is no data in the field.
-1 if the field is not found.

VARIANTS

```
number = xsm_n_num_occurs(field_name);
```

O_

variants that take a field number and occurrence number

SYNOPSIS

```
declare field_number       fixed binary(31);
declare occurrence         fixed binary(31);
call xsm_o_...(field_number, occurrence, ...);
```

DESCRIPTION

The o_ functions refer to data by field number and occurrence number. An occurrence is a slot within an array of fields in which data may be stored. Occurrences may be either on or off-screen. Since JAM treats an individual field as an array with one field, even a single non-scrolling field is considered to have one occurrence. The JAM library contains routines that allow you to manipulate individual occurrences during run-time.

If the occurrence is zero, the reference is always to the current contents of the specified field.

For the description of a particular function, look under the related function without o_ in its name. For example, xsm_o_amt_format is described under xsm_amt_format.

occur_no

get the current occurrence number

SYNOPSIS

```
declare occurrence        fixed binary(31);
occurrence = xsm_occur_no();
```

DESCRIPTION

This function returns the occurrence number of the field beneath the cursor. If the field is an element of a non–scrollable array, the occurrence number is the same as the field's element number. Likewise, the occurrence number of a single non–scrolling field is `1`.

RETURNS

0 if the cursor is not in a field.
Otherwise, the occurrence number.

RELATED FUNCTIONS

```
field_number = xsm_getcurno();
```

off_gofield

move the cursor into a field, offset from the left

SYNOPSIS

```
declare field_number      fixed binary(31);
declare offset            fixed binary(31);
declare status            fixed binary(31);
status = xsm_off_gofield(field_number, offset);
```

DESCRIPTION

This function moves the cursor into `field_number`, at position `offset` within the field's contents, regardless of the field's justification. The field's contents will be shifted if necessary to bring the appropriate piece onscreen.

If `offset` is larger than the field length (or the maximum length if the field is shiftable), the cursor will be placed in the rightmost position.

RETURNS

-1 if the field is not found.
0 otherwise.

VARIANTS

```
status = xsm_e_off_gofield(field_name, element, offset);
status = xsm_i_off_gofield(field_name, occurrence, offset);
status = xsm_n_off_gofield(field_name, offset);
status = xsm_o_off_gofield(field_number, occurrence, offset);
```

RELATED FUNCTIONS

```
offset = xsm_disp_off();
status = xsm_gofield(field_number);
offset = xsm_sh_off();
```

option

set a Screen Manager option

SYNOPSIS

```
declare option              fixed binary(31);
declare newval              fixed binary(31);
declare oldval              fixed binary(31);
oldval = xsm_option(option, newval);
```

DESCRIPTION

Use `xsm_option` to alter during run-time the default Screen Manager options defined in `smsetup.incl.pl1`. Possible options include, error window attributes, delayed write options, cursor display and zoom options. See the "Setup File" section in the *Configuration Guide* for a list of options and possible values. Use `xsm_keyoption` to alter the behavior of cursor control keys.

If you wish to simply inquire as to an option's current value, use the value NOCHANGE (defined in `smsetup.incl.pl1`) for `newval`.

This function replaces the following version 4.0 functions: `xsm_ch_emsgatt`, `xsm_ch_form_atts`, `xsm_ch_qmsgatt`, `xsm_ch_umsgatt`, `xsm_dw_options`, `xsm_er_options`, `xsm_fcase`, `xsm_fextension`, `xsm_ind_set`, `xsm_mp_options`, `xsm_mp_string`, `xsm_ok_options`, `xsm_stextatt`, and `xsm_zm_options`. They are included in your version 5.0 library only for backward compatibility. We strongly recommend that you do not use them in-the-future.

RETURNS

The old value for the specified option.
-1 if the option is out of range.

RELATED FUNCTIONS

```
oldval = xsm_keyoption(key, mode, newval);
```

oshift

shift a field by a given amount

SYNOPSIS

```
declare field_number       fixed binary(31);
declare offset             fixed binary(31);
declare return_value       fixed binary(31);
return_value = xsm_oshift(field_number, offset);
```

DESCRIPTION

This function shifts the contents of `field_number` by `offset` positions. If `offset` is negative, the contents are shifted right (data past the left-hand edge of the field become visible); otherwise, the contents are shifted left. Shifting indicators, if displayed, are adjusted accordingly.

The field may be shifted by fewer than `offset` positions if the maximum shifting width is reached with less shifting.

RETURNS

The number of positions actually shifted.
0 if the field is not found or is not shifting.

VARIANTS

```
return_value = xsm_n_oshift(field_name, offset);
```

pinquire
obtain value of a global strings

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare buffer              char(256) varying;
declare which               fixed binary(31);
buffer = xsm_pinquire(which);
```

DESCRIPTION

This function is used to obtain the current value of a global pointer variable. The mnemonics for `which` are defined in `smglobs.incl.pl1`. If you wish to modify a global string use `xsm_pset`.

Pointer values for `which` are defined in `smglobs.incl.pl1`. They are:

Mnemonic	*Meaning*
`P_YES`	The Y character for YES/NO field. This is returned as a three character string. The first character is the lowercase yes value, the second character is the uppercase yes value, and the third character is the null terminator.
`P_NO`	The N character for YES/NO field. This is returned as a three character string. The first character is the lowercase no value, the second character is the uppercase no value, and the third character is the null terminator.
`P_DECIMAL`	This is returned as a three character string. The first character is the user's decimal point marker, the second character is the operating system's decimal point marker, and the third character is the null terminator.
`P_FLDPTRS`	Pointer to an array of field structures. The implementation of these structures is very release dependent.
`P_TERM`	Returns the name JAM uses as the terminal identifier or the null string if not found.
`P_SPMASK`	Pointer to an memory-resident full size form containing all blanks.

Mnemonic	*Meaning*
P_USER	Pointer to developer-specified region of memory. This pointer is not set by JAM; it is set and maintained, if desired, by the application.
SP_NAME	Name of the active screen.
SP_STATLINE	Text of current status line.
SP_STATATTR	Attributes of current status line (pointer to array of unsigned short integers).
P_DICNAME	Name of data dictionary file.
V_	Any of the "V_" mnemonics defined in `smvideo.incl.pl1` may be passed to obtain various video related information.

In general, the objects pointed to by the pointers returned by `xsm_pinquire` have limited duration and should be used or copied quickly (except for P_USER, which is maintained by the application). The P_ pointers point to the actual objects within JAM. The SP_ pointers point to copies of the objects. Since the characteristics of these objects are implementation dependent, they may change in future releases of JAM. In no case (except P_USER) should the objects be modified directly through the pointers returned by `xsm_pinquire`. Use `xsm_pset` to modify selected objects).

RETURNS

If the argument corresponds to a global pointer variable, the value of that variable is returned.
0 otherwise.

RELATED FUNCTIONS

```
value = xsm_finquire(field_number, which);
value = xsm_gp_inquire(group_name, which);
value = xsm_iset(which, newval);
buffer = xsm_pset(which, newval);
```

protect

protect an array

SYNOPSIS

```
declare field_number        fixed binary(31);
declare mask                fixed binary(31);
declare status              fixed binary(31);

status = xsm_aprotect(field_number, mask);
status = xsm_aunprotect(field_number, mask);
status = xsm_protect(field_number);
status = xsm_unprotect(field_number);
status = xsm_1protect(field_number, mask);
status = xsm_1unprotect(field_number, mask);
```

DESCRIPTION

There are four types of protection associated with fields and arrays, any combination of which may be assigned: data entry, tabbing into, clearing, and validation. `xsm_protect` and `xsm_unprotect` always set and clear all four types of protection. The remaining protection functions set and clear any combination of protection, as specified by mask. The mnemonics for mask are defined in `smdefs.incl.pl1` and are listed below. Combinations may be specified by oring mnemonics together.

Mnemonic for mask	*Meaning*
EPROTECT	protect from data entry
TPROTECT	protect from tabbing into and from entering via any other key
CPROTECT	protect from clearing
VPROTECT	protect from validation
ALLPROTECT	protect from all of the above

Protection is associated an individual field (i.e. an element), and with an array as a whole. Therefore, all offscreen array occurrences always share the same level of protection,

while the onscreen occurrences have the levels of protection (possibly all different) associated with their host fields (i.e. elements). Since protection is associated with individual fields, and not with individual occurrences, deleting an occurrence with `xsm_doccur` will not scroll up the protection with the occurrences.

`xsm_protect`, `xsm_unprotect`, `xsm_1protect`, and `xsm_1unprotect` set and clear protection for individual fields. `xsm_aprotect` and `xsm_aunprotect` set and clear protection for all of the fields of an array, and for the array as a whole (the `field_number` may specify any field in the array). For example, unprotecting an array with `xsm_aunprotect` will undo protection done by `xsm_1protect`. A subsequent call to `xsm_1protect` will re-protect the specified field of the array, but can never affect the offscreen occurrences of the array.

Caution: It is generally safer to protect and unprotect arrays with `xsm_aprotect` and `xsm_aunprotect`, rather than with the field-oriented protection functions.

RETURNS

-1 if the field does not exist;
0 otherwise.

VARIANTS

```
status = xsm_n_protect(field_name);
status = xsm_e_protect(field_name, element);
status = xsm_n_unprotect(field_name);
status = xsm_e_unprotect(field_name, element);
status = xsm_n_1protect(field_name, mask);
status = xsm_e_1protect(field_name, element, mask);
status = xsm_n_1unprotect(field_name, mask);
status = xsm_e_1unprotect(field_name, element, mask);
status = xsm_n_aprotect(field_name, mask);
status = xsm_n_aunprotect(field_name, mask);
```

pset

Modify value of global strings

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare buffer              char(256) varying;
declare which               fixed binary(31);
declare newval              char(256) varying;
buffer = xsm_pset(which, newval);
```

DESCRIPTION

This function is used to modify the contents of a global string. The string you wish to change is specified by `which`. The value that you wish to change the variable to is specified by `newval`. If you wish only to get the value of a global string use `xsm_pinquire`.

The following values for `which`, defined in `smglobs.incl.pl1`, are available:

Mnemonic	*Meaning*
`P_YES`	The `Y` character for YES/NO field. This is specified by a three character string. The first character is the lowercase yes value, the second character is the uppercase yes value, and the third character is the null terminator.
`P_NO`	The `N` character for YES/NO field. This is specified by a three character string. The first character is the lowercase no value, the second character is the uppercase no value, and the third character is the null terminator.
`P_DECIMAL`	This is specified by a three character string. The first character is the user's decimal point marker, the second character is the operating system's decimal point marker, and the third character is the null terminator.

RETURNS

If `which` is one of the above, the old contents of the corresponding array are returned. 0 otherwise.

RELATED FUNCTIONS

```
value = xsm_iset(which, newval);
```

```
buffer = xsm_pinquire(which);
```

putfield

put a string into a field

SYNOPSIS

```
declare field_number        fixed binary(31);
declare data                char(256) varying;
declare status              fixed binary(31);
status = xsm_putfield(field_number, data);
```

DESCRIPTION

The string `data` is moved into the field specified by `field_number`. Strings that are too long will be truncated without warning, while strings shorter than the destination field are blank filled (to the left if the field is right justified, otherwise to the right). If `data` is a null string, then the field is cleared. This causes date and time fields that take system values to be refreshed.

This function sets the field's MDT bit to indicate that it has been modified, and clears its VALIDED bit to indicate that the field must be revalidated upon exit. `xsm_n_putfield` and `xsm_i_putfield` will store `data` in the LDB if the named field is not present in the screen. However, if the LDB item has a scope of 1 (constant), its contents will be unaltered and the function will return –1.

In variants that take `name` as an argument, `name` can be either the name of a field or a group. In the case of a group, the functions `xsm_select` and `xsm_deselect` should be used to change the group's value.

Notice that the order of arguments to this function is different from that of arguments to the related function `xsm_getfield`.

RETURNS

–1 if the field is not found; 0 otherwise.

VARIANTS

```
status = xsm_e_putfield(name, element, data);
status = xsm_i_putfield(name, occurrence, data);
status = xsm_n_putfield(name, data);
status = xsm_o_putfield(field_number, occurrence, data);
```

RELATED FUNCTIONS

```
status = xsm_deselect(group_name, group_occurrence);
```

```
length = xsm_getfield(buffer, field_number);
status = xsm_select(group_name, group_occurrence);
```

putjctrl

associate a control string with a key

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare key                fixed binary(31);
declare control_string     char(256) varying;
declare default            fixed binary(31);
declare status             fixed binary(31);
status = xsm_putjctrl(key, control_string, default);
```

DESCRIPTION

Each JAM screen contains a table of control strings associated with function keys. JAM also maintains a default table of keys and control strings, which take effect when the current screen has no control string for a function key you press. This table enables you to define system-wide actions for keys. It is initialized from SMINICTRL setup variables. See the section on setup in the Configuration Guide for further information.

This function associates `control_string` with `key` in one of the tables, replacing the control string previously associated with `key` (if there was one). If `default` is zero, the control string will be installed in the current screen, and will disappear when you exit the screen; otherwise, it will go into the system-wide default table. If `control_string` is empty, the existing control string, if any, will be deleted. If both screen and default control strings exist for a given key, deleting the control string for the screen will put the default control string into effect.

If you install a default control string for a key that is defined in the current screen, the definition in the screen will be used. Note also that JAM will not search the form or window stack for function key definitions; only the current screen and the default table are consulted. Mnemonics for `key` are in `smkeys.incl.pl1`. The syntax for control strings is defined in the Author's Guide.

RETURNS

-5 if insufficient memory is available; 0 otherwise.

pwrap

put text to a wordwrap field

SYNOPSIS

```
declare field_number      fixed binary(31);
declare text              char(256) varying;
declare status            fixed binary(31);
status = xsm_pwrap(field_number, text);
```

DESCRIPTION

This function copies `text` to a wordwrap field specified by `field_number`. Wraps occur at the end of words. The last character of every line is a space. If a word is longer than one less than the length of the field, the word is broken one character short of the end of the field, a space is appended, and the remainder of the word wraps to the next line.

The variant `xsm_o_pwrap` copies the text into an array beginning at the specified occurrence.

Warning: If you attempt to copy data that is too large for the wordwrap field to hold, `xsm_pwrap` will truncate the excess text.

RETURNS

-1 if the field number is invalid.
-2 if the text was truncated because it was too long for the field.
0 otherwise.

VARIANTS

```
status = xsm_o_pwrap(field_number, occurrence, text);
```

RELATED FUNCTIONS

```
length = xsm_gwrap(buffer, field_number, buffer_length);
```

query_msg

display a question, and return a yes or no answer

SYNOPSIS

```
declare message              char(256) varying;
declare reply                fixed binary(31);
reply = xsm_query_msg(message);
```

DESCRIPTION

The message is displayed on the status line, until you type a yes or a no key. A yes key is the first letter of the SM_YES entry in the message file (or the XMIT key), and a no key is the first letter of the SM_NO entry (or the EXIT key); case is ignored. At that point, this function returns the lower case letter as defined in the message file to its caller.

All keys other than yes and no keys are ignored.

RETURNS

Lower-case ASCII 'y' or 'n', according to the response.

RELATED FUNCTIONS

```
call xsm_d_msg_line(message, display_attribute);
status = xsm_is_no(field_number);
status = xsm_is_yes(field_number);
```

qui_msg

display a message preceded by a constant tag, and reset the message line

SYNOPSIS

```
declare message            char(256) varying;
call xsm_qui_msg(message);
```

DESCRIPTION

This function prepends a tag (normally "ERROR:") to `message`, and displays the whole on the status line (or in a window if it is too long). The tag may be altered by changing the `SM_ERROR` entry in the message file. The message remains visible until the operator presses a key. Refer to the description of setup in the Configuration Guide for an exact description of error message acknowledgement. If the message is longer than the status line, it will be displayed in a window instead. If the cursor position display has been turned on (see `xsm_c_vis`), the end of the status line will contain the cursor's current row and column. If the message text would overlap that area of the status line, it will be displayed in a window instead.

This function is identical to `xsm_quiet_err`, except that it does not turn the cursor on. It is similar to `xsm_emsg`, which does not prepend a tag.

Several *percent escapes* provide control over the content and presentation of status messages. See `xsm_emsg` for details.

RELATED FUNCTIONS

```
call xsm_emsg(message);
call xsm_err_reset(message);
oldval = xsm_option(option, newval);
call xsm_quiet_err(message);
```

quiet_err

display error message preceded by a constant tag, and reset the status line

SYNOPSIS

```
declare message          char(256) varying;
call xsm_quiet_err(message);
```

DESCRIPTION

This function prepends a tag (normally "ERROR") to `message`, turns the cursor on, and displays the whole message on the status line (or in a window if it is too long). This function is identical to `xsm_qui_msg`, except that it turns the cursor on. It is similar to `xsm_err_reset`, which does not prepend a tag. Refer to `xsm_emsg` for an explanation of how to change display attributes and insert function key names within a message.

RELATED FUNCTIONS

```
call xsm_emsg(message);
call xsm_err_reset(message);
oldval = xsm_option(option, newval);
call xsm_qui_msg(message);
```

rd_part

read part of a data structure to the current screen

SYNOPSIS

```
declare screen_struct      bit(0);
declare first_field        fixed binary(31);
declare last_field         fixed binary(31);
call xsm_rd_part(screen_struct, first_field, last_field);
```

DESCRIPTION

This function copies data from a structure to all fields between `first_field` and `last_field` within the current screen, converting individual members as appropriate. An array and its scrolling occurrences will be copied only if the *first* element falls between `first_field` and `last_field`. This routine is commonly used with `xsm_wrt_part`, which writes part of the screen to a structure. If you wish to read information into the entire screen, use `xsm_rdstruct`. To read information into a data dictionary record, use `xsm_rrecord`. Use `xsm_putfield` to write a string to an individual field.

A data structure named `screen` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gPL1 screen.jam
```

Each member of the structure is a field of the type specifed in the Screen Editor. If you specify the type `omit`, data will not be written into the field. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other structure declaration. You can ignore the items that represent fields which do not fall within the bounds of the specifed fields. However, the structure definition must contain all of the fields on the screen. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`.

The arguments that represent the range of fields to be copied, `first_field` and `last_field` are passed as field numbers.

The structure may be initialized with `xsm_wrt_part` or with data from elsewhere. Structure members within the specified range which will not be initialized prior to calling `xsm_rd_part` must be zeroed-out or you risk crashing your application when garbage is read into the screen.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
call xsm_rd_struct(screen_struct, byte_count);
call xsm_rrecord(structure_ptr, record_name, byte_count);
call xsm_wrt_part(screen_struct, first_field, last_field);
```

rdstruct

read data from a structure to the screen

SYNOPSIS

```
declare screen_struct      bit(0);
declare byte_count         fixed binary(31);
call xsm_rd_struct(screen_struct, byte_count);
```

DESCRIPTION

This function copies data from a structure to the current screen, converting individual members as appropriate. It is commonly used with `xsm_wrtstruct`, which writes data from fields on the current screen to a structure. If you wish to read information into a group of consecutively numbered fields, use `xsm_rd_part`. To read information from a data dictionary record, use `xsm_rrecord`. Use `xsm_putfield` to write a string to an individual field.

A data structure named `screen` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gPL1 screen.jam
```

Each member of the structure is a field of the type specifed in the Screen Editor. If you specify the type `omit`, data will not be written into the field. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other structure declaration. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`.

The argument `byte_count` is an integer variable. `xsm_rdstruct` will store in `byte_count` the number of bytes copied from the structure.

The structure may be initialized with `xsm_wrtstruct` or with data from elsewhere. Members within the structure that will not be initialized prior to calling `xsm_rdstruct` must be zeroed-out or you risk crashing your application when garbage is read into the screen.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
```

```
call xsm_rd_part(screen_struct, first_field, last_field);
call xsm_rrecord(structure_ptr, record_name, byte_count);
call xsm_wrtstruct(screen_struct, byte_count);
```

rescreen

refresh the data displayed on the screen

SYNOPSIS

```
call xsm_rescreen();
```

DESCRIPTION

This function repaints the entire display from JAM's internal screen and attribute buffers. Anything written to the screen by means other than JAM library functions will be erased. This function is normally bound to the RESCREEN key and executed automatically within `xsm_getkey`.

You may need to use this function after doing screen I/O with the flag `xsm_do_not_display` turned on, or after escaping from an JAM application to another program (see `xsm_leave`). If all you want is to force writes to the display, use `xsm_flush`.

RELATED FUNCTIONS

```
call xsm_flush();
call xsm_return();
```

resetcrt

reset the terminal to operating system default state

SYNOPSIS

```
call xsm_resetcrt();
call xsm_jresetcrt();
call xsm_jxresetcrt();
```

DESCRIPTION

The function `xsm_resetcrt` is generally used only when you are writing your own Executive. It resets terminal characteristics to the operating system's normal state. Be sure to call `xsm_resetcrt` be called when leaving the Screen Manager environment (before program exit).

All the memory associated with the display and open screens is freed However, the buffers holding the message file, key translation file, etc. are not released. A subsequent call to `xsm_initcrt` will find them in place. Then `xsm_resetcrt` clears the screen and turns on the cursor, transmits the RESET sequence defined in the video file, and resets the operating system channel.

The JAM Executive calls `xsm_resetcrt` via `xsm_jresetcrt` (or via `xsm_jxresetcrt` in the case of an authoring executable) automatically as part of its exit processing. It should not be called by application programs except in case of abnormal termination.

RELATED FUNCTIONS

```
call xsm_cancel();
call xsm_leave();
```

resize

notify **JAM** of a change in the display size

SYNOPSIS

```
declare rows                fixed binary(31);
declare columns             fixed binary(31);
declare status              fixed binary(31);
status = xsm_resize(rows, columns);
```

DESCRIPTION

This function enables you to change the size of the display used by JAM from the default defined by the LINES and COLMS entries in the video file. It makes it possible to use a single video file in a windowing environment. Applications can be run in different sized windows with each application setting its display size at run time. It can also be used for switching between normal and compressed modes (e.g. 80 and 132 columns on VT100-compatible terminals).

If the specified rectangle is larger than the physical display, the results will be unpredictable. You may specify at most 255 rows or columns.

This function clears the physical and logical screens; any displayed forms or windows, together with data entered on them, are lost.

RETURNS

-1 if a parameter was less than 0 or greater than 255.
0 if successful.
Program exit on memory allocation failure.

return

prepare for return to **JAM** application

SYNOPSIS

```
call xsm_return();
```

DESCRIPTION

This routine should be called upon returning to a JAM application after a temporary exit.

It sets up the operating system channel and initializes the display using the `SETUP` string from the video file. It does *not* restore the screen to the state it was in before `xsm_leave` was called. Use `xsm_rescreen` to accomplish that, if desired.

RELATED FUNCTIONS

```
call xsm_leave();
call xsm_resetcrt();
```

rmformlist

empty the memory–resident form list

SYNOPSIS

```
call xsm_rmformlist;
```

DESCRIPTION

This function erases the memory–resident form list established by `xsm_formlist`, and releases the memory used to hold it. It does not release any of the memory–resident JPL modules, key sets, or screens themselves. Calling this function will prevent `xsm_r_window`, `xsm_r_keyset`, `xsm_jplcall`, and related functions from finding memory–resident objects.

RELATED FUNCTIONS

```
status = xsm_formlist(name, address);
```

rrecord

read data from a structure to a data dictionary record

SYNOPSIS

```
declare structure_ptr      bit(0);
declare record_name        char(256) varying;
declare byte_count         fixed binary(31);
call xsm_rrecord(structure_ptr, record_name, byte_count);
```

DESCRIPTION

This function reads data from a PL/1 structure into fields on the current screen that are part of a common data dictionary record. If a field is not on the current screen then the data is written to the LDB. This routine is commonly used with `xsm_wrecord`, which writes data from a data dictionary record to a PL/1 structure. If you wish to read data into all of the fields within the current screen, use `xsm_rdstruct`. To copy data to a group of consecutively numbered fields, use `xsm_rd_part`. Use `xsm_putfield` to write a string to an individual field.

A data structure named `record` can be created from the data dictionary file `data.dic` via the `dd2struct` utility as follows:

```
dd2struct -gPL1 data.dic
```

Each structure member is a field within a data dictionary record that is of the type specified in the Screen Editor. Data will be written into the field onscreen even if the `omit` type is specified. See "Data Type" in the Author's Guide and `dd2struct` in the Utilities Guide for further information.

Once created, the declarations may be treated exactly like any other structure declarations. The argument `struct_ptr` is the address of a variable of one of the structure types generated by `dd2struct`. The argument `record_name` is the name of the data dictionary record from which the structure was created.

The argument `byte_count` is a pointer to an integer. Upon return from `xsm_rrecord`, the value contained in the integer will be the number of bytes or characters read from the structure. The value will be 0 if an error occurred.

The structure may be initialized with `xsm_wrecord` or with data from elsewhere. Members within the structure that will not be initialized prior to calling `xsm_rrecord` must be zeroed-out or you risk crashing your application when garbage is read into the screen or the LDB.

Remember, you must update the structure declaration whenever you alter the data dictionary from which it was generated.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
call xsm_rd_part(screen_struct, first_field, last_field);
call xsm_rd_struct(screen_struct, byte_count);
call xsm_wrecord(structure_ptr, record_name, byte_count);
```

rscroll

scroll an array

SYNOPSIS

```
declare field_number       fixed binary(31);
declare req_scroll         fixed binary(31);
declare lines              fixed binary(31);
lines = xsm_rscroll(field_number, req_scroll);
```

DESCRIPTION

This function scrolls an array along with any synchronized arrays by `req_scroll` occurrences. If `req_scroll` is positive, the array scrolls down (towards the bottom of the data); otherwise, it scrolls up.

The function returns the actual amount scrolled. This could be the amount requested, or a smaller value if the requested amount would bring the array past its beginning or end. If 0 is returned it means that the array was at its beginning or end, or an error occurred. Negative numbers indicate scrolling up occurred.

RETURNS

The actual amount scrolled. Positive numbers indicate downward scrolling while negative numbers mean upward scrolling.
0 if no scrolling or error.

VARIANTS

```
lines = xsm_n_rscroll(field_name, req_scroll);
```

RELATED FUNCTIONS

```
status = xsm_ascroll(field_number, occurrence);
status = xsm_t_scroll(field_number);
```

s_val

validate the current screen

SYNOPSIS

```
declare status              fixed binary(31);
status = xsm_s_val();
```

DESCRIPTION

This function validates each field and occurrence, whether on or offscreen, that is not protected from validation (VPROTECT). It is called automatically from `xsm_input` when the TRANSMIT key is hit while in data entry mode. `xsm_sval` also validates groups.

When the first element of a scrolling array is encountered, earlier offscreen occurrences are validated first. When the last element of a scrolling array is encountered, later offscreen occurrences are validated immediately after that element.

If synchronized arrays exist, the following occurs. When an offscreen occurrence is validated, the corresponding occurrences from synchronized arrays are validated as well. Synchronized array are validated in order according to their base field number. The offscreen occurrences *preceding* the synchronized arrays are validated before the first onscreen occurrence of the first (lowest base field number) of the synchronized arrays. Similarly, the offscreen occurrences *following* the arrays are validated immediately after the last onscreen occurrence of the last (highest base field number) array.

Validation	*Skip if valid*	*Skip if empty*
required	y	n
must fill	y	y
regular expression	y	y
range	y	y
check-digit	y	y
date or time	y	y
table lookup	y	y
currency format	y	n*
math expresssion	n	n

Validation	*Skip if valid*	*Skip if empty*
field validation	n	n
JPL function	n	n

* The currency format edit contains a skip–if–empty flag; see the Author's Guide.

If you need to force a skip–if–empty validation, make the field required. A field with embedded punctuation must contain at least one non–blank non–punctuation character in order to be considered non–empty; otherwise any non blank character makes the field non–empty.

If an occurrence fails validation, the cursor is positioned to it and an error message displayed. If the occurrence was offscreen, its the array is first scrolled to bring it onscreen. This routine returns at the first error; any fields past will not be validated.

RETURNS

–1 if any field fails validation.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_fval(field_number);
```

sc_max

alter the maximum number of occurrences allowed in a scrollable array

SYNOPSIS

```
declare field_number        fixed binary(31);
declare new_max             fixed binary(31);
declare actual_max          fixed binary(31);
actual_max = xsm_sc_max(field_number, new_max);
```

DESCRIPTION

This function changes the maximum number of occurrences allowed in `field_number`, and in all synchronized arrays. The original maximum is set when the screen is created. If the desired new maximum is less than the highest numbered occurrence that contains data, the new maximum will be set to the number of that occurrence (i.e., the value returned by `xsm_num_occurs`). The maximum can decrease only to a value between the highest numbered occurrence containing data and the previous maximum. It can never be less than the number of elements in the array.

RETURNS

The actual new maximum (see above).
0 if the desired maximum is invalid, or if the array is not scrollable.

VARIANTS

```
actual_max = xsm_n_sc_max(field_name, new_max);
```

RELATED FUNCTIONS

```
maximum = xsm_max_occur(field_number);
number = xsm_num_occurs(field_number);
```

sdtime

get formatted system date and time

SYNOPSIS

```
declare buffer              char(256) varying;
declare format              char(256) varying;
buffer = xsm_sdtime(format);
```

DESCRIPTION

This function gets the current date and/or time from the operating system and returns it in the form specified by `format`.

`format` is a string beginning with `y` or `n` followed by any combination of date/time tokens and literal text. `y` indicates a 12–hour clock; `n` (or any other character) indicates a 24–hour clock. This character must be given, even if the format does not include time tokens. The tokens are described in the table below. These tokens are case–sensitive.

Unit	*Description*	*Token*
Year	4 digit (e.g., 1990)	`%4y`
	2 digit (e.g., 90)	`%2y`
Month	1 or 2 digit (1 – 12)	`%m`
	2 digit (01 – 12)	`%0m`
	full name (e.g., January)	`%*m`
	3 character name (e.g., Jan)	`%3m`
Day	1 or 2 digit (1 – 31)	`%d`
	2 digit (01 – 31)	`%0d`
Day of the Week	full name (e.g. Sunday)	`%*d`
	3 character name (e.g., Sun)	`%3d`
Day of the Year	digit (1 – 365)	`%+d`

Unit	*Description*	*Token*
Hour	1 or 2 digit (1 – 12 or 1 – 24)	`%h`
	2 digit (01 –12 or 01 –24)	`%0h`
Minute	1 or 2 digit (1 – 59)	`%M`
	2 digit (01 – 59)	`%0M`
Second	1 or 2 digit (1 – 59)	`%s`
	2 digit (01 – 59)	`%0s`
AM or PM	for use with a 12–hour clock	`%p`
Literal Percent	use `%` as a literal character	`%%`
Ten Default Formats	`SM_0DEF_DTIME`	`%0f`
(from the message file)	`SM_1DEF_DTIME`	`%1f`
	...	...
	`SM_9DEF_DTIME`	`%09f`

At runtime, JAM strips off the first character of `format`. If the character is `y`, it uses a 12–hour clock; else it uses the default 24–hour clock. Next it examines the rest of `format`, replacing any tokens with the appropriate values. All other characters are used literally. Therefore, be sure to put a `y` or an `n` (or perhaps a blank) at the beginning of `format`. If you do not, JAM strips off the first token's percent sign and it treats the rest of the token as literal text.

You may also retrieve a date/time format from a field using `xsm_edit_ptr`.

The text for day and month names, AM and PM, as well as the tokens for the ten default formats, are all stored in the message file. These entries may be modified. See the Configuration Guide for details.

Note: This function replaces Release 4's `xsm_sdate` and `xsm_stime` function.

RETURNS

The current date/time in the specified format.
Empty if `format` is invalid.

RELATED FUNCTIONS

```
status = xsm_calc(field_number, occurrence, expression);
```

select

select a checklist or radio button occurrence

SYNOPSIS

```
declare group_name        char(256) varying;
declare group_occurrence  fixed binary(31);
declare status            fixed binary(31);
status = xsm_select(group_name, group_occurrence);
```

DESCRIPTION

This function allows you to select a specific occurrence within a checklist or radio button. The group name and occurrence number are used to reference the desired selection.

Use `xsm_deselect` to deselect a checklist occurrence.

Selecting a radio button occurrence automatically causes the currently selected radio button to be deselected, because exactly one occurrence in a radio button group must be selected at all times. See the Author's Guide for a more detailed discussion of groups.

Use `xsm_isselected` to check whether or not a particular radio button or checklist occurrence is currently selected.

RETURNS

-1 arguments do not reference a checklist or radio button occurrence.
0 occurrence not previously selected.
1 occurrence previously selected.

RELATED FUNCTIONS

```
status = xsm_deselect(group_name, group_occurrence);
status = xsm_isselected(group_name, group_occurrence);
```

setbkstat

set background text for status line

SYNOPSIS

```
declare message           char(256) varying;
declare display_attribute fixed binary(31);
call xsm_setbkstat(message, display_attribute);
```

DESCRIPTION

The `message` is saved, to be shown on the status line whenever there is no higher priority message to be displayed. The highest priority messages are those passed to `xsm_d_msg_line`, `xsm_err_reset`, `xsm_quiet_err`, or `xsm_query_msg`; the next highest are those attached to a field by means of the status text option (see the JAM Author's Guide). Background status text has lowest priority.

Possible values for the `display_attribute` argument are defined in the header file `smdefs.incl.pll`, as shown in the table below:

Foreground Attributes	*Background Attributes*
BLANK	B_HILIGHT
REVERSE	
UNDERLN	
BLINK	
HILIGHT	
STANDOUT	
DIM	
ACS (alternate character set)	
Foreground Colors	*Background Colors*
BLACK	B_BLACK
BLUE	B_BLUE
GREEN	B_GREEN
CYAN	B_CYAN

Foreground Colors	*Background Colors*
RED	B_RED
MAGENTA	B_MAGENTA
YELLOW	B_YELLOW
WHITE	B_WHITE

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

`xsm_setstatus` sets the background status to an alternating ready/wait flag; you should turn that feature off before calling this routine.

Refer to `xsm_d_msg_line` for an explanation of how to embed attribute changes and function key names into your message.

RELATED FUNCTIONS

```
call xsm_d_msg_line(message, display_attribute);
call xsm_setstatus(mode);
```

setstatus

turn alternating background status message on or off

SYNOPSIS

```
declare mode                fixed binary(31);
call xsm_setstatus(mode);
```

DESCRIPTION

If `mode` is non–zero, alternating status flags are turned on. After this call, one message (normally `Ready`) is displayed on the status line while JAM is waiting for input, and another (normally `Wait`) when it is not. If `mode` is zero, the messages are turned off.

The status flags will be replaced temporarily by messages passed to `xsm_err_reset` or a related routine. They will overwrite messages posted with `xsm_d_msg_line` or `xsm_setbkstat`.

The alternating messages are stored in the message file as `SM_READY` and `SM_WAIT`, and can be changed there. Attribute changes and function key names can be embedded in the messages; refer to `xsm_d_msg_line` for instructions.

RELATED FUNCTIONS

```
call xsm_setbkstat(message, display_attribute);
```

sh_off

determine the cursor location relative to the start of a shifting field

SYNOPSIS

```
declare offset              fixed binary(31);
offset = xsm_sh_off();
```

DESCRIPTION

Returns the difference between the start of data in a shiftable field and the current cursor location. If the current field is not shiftable, it returns the difference between the leftmost column of the field and the current cursor location, like `xsm_disp_off`.

RETURNS

The difference between the current cursor position and the start of shiftable data in the current field.
-1 if the cursor is not in a field.

RELATED FUNCTIONS

```
offset = xsm_disp_off();
```

shrink_to_fit

remove trailing empty array elements and shrink screen

SYNOPSIS

```
call xsm_shrink_to_fit();
```

DESCRIPTION

Use this routine to dynamically downsize the current screen when you don't know how many elements of an array are going to be populated with data at run time. This routine removes all trailing elements in all arrays on screen and then shrinks the screen to a size just large enough to accommodate the displayed data. If no data is placed in the array, the entire array will be removed. Only the currently displayed copy of the screen in memory is altered.

This routine only downsizes the array and screen. It will not enlarge an array or screen that is too small to hold the information, so be sure to create, within the Screen Editor, an array and screen that can hold the largest amount of data that you plan on inserting.

sibling

define the current window as being or not being a sibling window

SYNOPSIS

```
declare should_it_be       fixed binary(31);
call xsm_sibling(should_it_be);
```

DESCRIPTION

Users may switch between the active window and all siblings of that window while they are in viewport mode. Sibling windows must be next to each other on the window stack. When a window is defined as a sibling, then it and the window immediately beneath it on the window stack are considered to be siblings of one another. The user enters viewport mode when either the VWPT (viewport) logical key is pressed or when the application program makes a call to `xsm_winsize`.

Use this function to define whether or not the current window is defined as sibling To change the current sibling status of a window assign `should_it_be` to:

0	No, it is not a sibling window.
1	Yes, it is a sibling window.

To understand how sibling windows work, imagine you have a stack of three windows: `window_top`, `window_middle`, and `window_bottom`. To make `window_top` and `window_middle` siblings of each other, define `window_top` as a sibling window. They are now considered siblings of each other. You can then add a third sibling to the pair, by defining `window_middle` as a sibling window. This results in `window_middle` and `window_bottom` becoming siblings of one another and consequently, `window_top` and `window_bottom` are also siblings of each other. There is no limit to the number of siblings window you may chain together in this fashion, as long as the windows are adjacent to each other on the stack.

If you wish to bring a different window to the top of the stack, use `xsm_wselect`. To get the number of windows currently in the window stack use `xsm_wcount`.

The base form can be a sibling of the windows adjacent to it.

RELATED FUNCTIONS

```
return_value = xsm_wcount();
status = xsm_winsize();
```

```
return_value = xsm_wselect(window_number);
```

size_of_array

get the number of elements

SYNOPSIS

```
declare field_number        fixed binary(31);
declare size                fixed binary(31);
size = xsm_size_of_array(field_number);
```

DESCRIPTION

This function returns the number of elements in the array containing `field_number`. Elements are the onscreen portion of an array. An array always has at least one element.

RETURNS

0 if the field designation is invalid.
1 if the field is not an array.
The number of elements in the array otherwise.

VARIANTS

```
size = xsm_n_size_of_array(field_name);
```

RELATED FUNCTIONS

```
maximum = xsm_max_occur(field_number);
```

skinq

obtain soft key information by position

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

%include 'smkeys.incl.pl1';

declare scope              fixed binary(31);
declare row                fixed binary(31);
declare softkey            fixed binary(31);
declare value              fixed binary(31);
declare display_attribute  fixed binary(31);
declare label1             char(256) varying;
declare label2             char(256) varying;
declare status             fixed binary(31);
size = xsm_skinq(scope, row, softkey, value, display_attribute,
         label1, label2);
```

DESCRIPTION

Use this routine to obtain the value, attributes, and label of a soft key contained in a keyset currently in memory, given a soft key's position within a keyset.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Mnemonics for `scope` are defined in `smsoftk.incl.pl1`. For a more detailed explanation of scope see the Keyset chapter of the Programmer's Guide.

The logical value of the specified soft key is placed in `value`. This will be a number that corresponds to a mnemonic defined in `smkeys.incl.pl1`. A value of 0 means the key is inactive.

The attributes (color, blinking etc...) of the label will be placed in `display_attribute`. The attribute should be one of the mnemonics listed in `smdefs.incl.pl1`.

The first and second row labels are placed in `label1` and `label2` respectively. You should pre-allocate at least nine elements for `label1` and `label2` buffers (eight for the label characters and one for the null character).

If you do not desire information about one or more of these parameters you may assign the parameters the null pointer.

If you want general information about a keyset, see `xsm_ksinq`. If you want the scope of the current keyset, use `xsm_kscscope`.

WARNING: This routine can not be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs JAM automatically breaks the rows to position them correctly on the monitor. This means that you will not be able to reliably reference a particular soft key by its row and position. Instead, use `xsm_skvinq`.

RETURNS

0 if information has been returned.
-1 if there is no active keyset for the given scope.
-2 for an invalid scope.
-3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
scope = xsm_kscscope();
status = xsm_ksinq(scope, number_keys, number_rows,
         current_row, maximum_len, keyset_name);
status = xsm_skvinq(scope, value, occurrence, attribute,
         label1, label2);
```

skmark

mark or unmark a soft key label by position

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

declare scope              fixed binary(31);
declare row                fixed binary(31);
declare softkey            fixed binary(31);
declare mark               fixed binary(31);
declare status             fixed binary(31);
status = xsm_skmark(scope, row, softkey, mark);
```

DESCRIPTION

Use this routine to mark or unmark a soft key label in an open keyset. The mark is made in the last position of the first label.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.pl1`. The argument `row` is the row number in which the desired `softkey` resides. Rows are counted from top to bottom, beginning with `1`. The argument `softkey` is the position number within `row` of the desired soft key. Positions are numbered left to right, beginning with `1`.

The argument `mark` may be any single ASCII character. An asterisk (*) is the most commonly used mark. To unmark the key use the space character (' ') for `mark`.

The marking or unmarking of a soft key is often done to indicate a selection on a function key that toggles between two options.

WARNING: This routine can not be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs JAM automatically breaks the rows to position them correctly on the monitor. This means that you will not be able to reliably reference a particular soft key by its row and position. Instead, use `xsm_skvmark`.

RETURNS

0 if the marking was successful.
-1 if there is no keyset of the specified scope.
-2 if the scope is out of range.
-3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
status = xsm_skvmark(scope, value, occurrence, mark);
```

skset

set characteristics of a soft key by position

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

%include 'smkeys.incl.pl1';

%include 'smkeys.incl.pl1';

declare scope              fixed binary(31);
declare row                fixed binary(31);
declare softkey            fixed binary(31);
declare value              fixed binary(31);
declare attribute          fixed binary(31);
declare label1             char(256) varying;
declare label2             char(256) varying;
declare status             fixed binary(31);
status = xsm_skset(scope, row, softkey, value, attribute,
         label1, label2);
```

DESCRIPTION

This routine can be used to modify a soft key's scope, value, attribute, or label of any currently open keysets. You may modify one or more of these specifications with each call of `xsm_skset`.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.pl1`. The argument `row` is the row number in which the desired `softkey` resides. Rows are counted from top to bottom, beginning with `1`. The argument `softkey` is the position number within `row` of the desired soft key. Positions are numbered left to right, beginning with `1`.

The `value` refers to the logical key name to be assigned to the soft key. Available mnemonics are defined in `smkeys.incl.pl1`. If you do not want to change the logical name, assign `-1` to `value`.

The `attribute` (color, blinking, etc.) is specified by using mnemonics listed in `smdefs.incl.pl1`. If you do not want to change `attribute`, assign it `0`. (Note: If you set both the background and foreground to black, `xsm_skset` will set the foreground to white, provided that the terminal supports background color.)

The variables `label1` and `label2` are the first and second lines of the labels respectively. If you do not wish to change one of the labels, assign it the null pointer.

WARNING: This routine can not be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs JAM automatically breaks the rows to position them correctly on the monitor. This means that you will not be able to reliably reference a particular soft key by its row and position. Instead, use `xsm_skvset`.

RETURNS

0 if no error has occurred.
-1 if there is no active keyset for the given scope.
-2 for an invalid scope.
-3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
status = xsm_skvset(scope, value, occurrence, newval,
         attribute, label1, label2);
```

skvinq

obtain soft key information by value

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

%include 'smkeys.incl.pl1';

declare scope              fixed binary(31);
declare value              fixed binary(31);
declare occurrence         fixed binary(31);
declare attribute          fixed binary(31);
declare label1             char(256) varying;
declare label2             char(256) varying;
declare status             fixed binary(31);
status = xsm_skvinq(scope, value, occurrence, attribute,
         label1, label2);
```

DESCRIPTION

Use this routine to obtain the label text and attributes of a soft key contained in a keyset currently in memory, given the soft key's value. It can be used when the terminal has a different number of keys than the keyset was designed for.

The soft key is referenced by the keyset it belongs to, its value, and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.pl1`. The `value` of the soft key is one of the mnemonic defined in `smkeys.incl.pl1`. The argument `occurrence` specifies which occurrence of a key with the specified value is desired (in case of duplicates).

The attributes (color, blinking etc . . .) of the label will be placed in `attribute`. The value of the attributes correspond to a mnemonic, or some combination of ored mnemonics listed in `smdefs.incl.pl1`.

The first and second row labels are placed in `label1` and `label2` respectively. You should pre-allocate at least nine elements for `label1` and `label2` buffers (eight for the label characters and one for the null character).

If you do not desire information about one or more of these parameters you may assign the parameters the null pointer.

For general information about a keyset, see `xsm_ksinq`. If you want the scope of the current keyset, use `xsm_kscscope`.

RETURNS

0 if information has been returned.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
size = xsm_skinq(scope, row, softkey, value, display_attribute,
        label1, label2);
```

skvmark

mark a soft key by value

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

%include 'smkeys.incl.pl1';

declare scope              fixed binary(31);
declare value              fixed binary(31);
declare occurrence         fixed binary(31);
declare mark               fixed binary(31);
declare status             fixed binary(31);
status = xsm_skvmark(scope, value, occurrence, mark);
```

DESCRIPTION

Use this routine to mark or unmark a soft key label in an open keyset. The mark is made in the last position of the first label.

The soft key is referenced by the keyset it belongs to, its value and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.pl1`. The `value` of the soft key is one of the mnemonic defined in `smkeys.incl.pl1`. The argument `occurrence` is the nth time that `value` appears in the keyset. If you wish to mark all occurrences of `value` assign 0 to `occurrence`.

The argument `mark` may be any single ASCII character. An asterisks (*) is the most commonly used mark. To unmark the key use the space character (' ') for `mark`.

The marking or unmarking of a soft key is often done to indicate a selection on a function key that toggles between two options.

RETURNS

0 if the mark was successful.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
status = xsm_skmark(scope, row, softkey, mark);
```

skvset

set characteristics of a soft key by value

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

%include 'smkeys.incl.pl1';

declare scope              fixed binary(31);
declare value              fixed binary(31);
declare occurrence         fixed binary(31);
declare newval             fixed binary(31);
declare attribute          fixed binary(31);
declare label1             char(256) varying;
declare label2             char(256) varying;
declare status             fixed binary(31);
status = xsm_skvset(scope, value, occurrence, newval,
         attribute, label1, label2);
```

DESCRIPTION

This routine can be used to modify the scope, value, attribute, or label of a soft key within a currently open keyset. You may modify one or more of these specifications with each call of `xsm_skset`.

The soft key is referenced by the keyset it belongs to, its value and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.pl1`. The `value` of the soft key is one of the mnemonic defined in `smkeys.incl.pl1`. The argument `occurrence` is the nth time that `value` appears in the keyset. If you wish to change all occurrences of `value` assign 0 to `occurrence`.

The value of `newvalue` refers to the logical key name to be assigned to the soft key. Available mnemonics are defined in `smkeys.incl.pl1`. If you do want to change the logical name, assign `-1` to `value`.

The `attribute` (color, blinking, etc.) is specified by using mnemonics listed in `smdefs.incl.pl1`. If you do not want to change `attribute`, assign it `0`. (Note: If you set both the background and foreground to black, `xsm_skset` will set the foreground to white, provided that the terminal supports background color.)

The variables `label1` and `label2` are the first and second lines of the labels respectively. If you do not wish to change one of the labels, assign it the null pointer.

RETURNS

0 if no error occurred
–1 if there is no active keyset for the given scope
–2 for an invalid scope
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
status = xsm_skset(scope, row, softkey, value, attribute,
         label1, label2);
```

strip_amt_ptr

strip amount editing characters from a string

SYNOPSIS

```
declare field_number       fixed binary(31);
declare inbuf              char(256) varying;
declare outbuf             char(256) varying;
outbuf = xsm_strip_amt_ptr(field_number, inbuf);
```

DESCRIPTION

Strips all non–digit characters from the string, except for an optional leading minus sign and decimal point. If `inbuf` is not empty, `field_number` is ignored and the passed string is processed in place.

If `inbuf` is empty, the contents of `field_number` are used.

RETURNS

The stripped text,
0 if `inbuf` is empty and the field number is invalid.

RELATED FUNCTIONS

```
status = xsm_amt_format(field_number, buffer);
value = xsm_dblval(field_number);
```

submenu_close

close the current submenu

SYNOPSIS

```
declare status              fixed binary(31);
status = xsm_submenu_close();
```

DESCRIPTION

Submenus are ordinarily closed before `xsm_input` returns. It may, however, be told to leave them open by using the OK_LEAVEOPEN option, either in the setup file or via `xsm_option`. See the *Configuration Guide* for details. Regardless of how this option is set, submenus are automatically closed whenever the underlying window is closed with `xsm_close_window`.

This function, then, is needed only when all of the following conditions are true.

1. OK_LEAVEOPEN is in use.
2. The submenu is no longer needed.
3. Access is needed to the underlying window.

RETURNS

-1 if there is no submenu currently open.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_close_window();
```

svscreen

register a list of screens on the save list

SYNOPSIS

```
declare screen_list        char(256) varying;
declare count              fixed binary(31);
declare status             fixed binary(31);
status = xsm_svscreen(screen_list, count);
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. The number of screens to be added is given by `count`. You may add screens to the list anywhere within your code, however the screen is not actually placed in memory until it is closed for the first time. This means that the time saving factor only comes into play in subsequent openings of the screen. Any data entered into a screen will not be saved until the screen is closed.

Screens are removed from the list with `xsm_unsvscreen`. You can check to see if a screen is on the save list with `xsm_issv`. Checking the list prior to calling `xsm_svscreen`, however, is not crucial as any attempt to add a screen that is already on the list will have no effect.

This routine saves processing time at the expense of memory. It is best suited for use with screens that both require large amounts of data to be read in from elsewhere (databases, other files, etc.) and do not allow the user to enter data. For instance, if you have a help screen that needs to be populated by a data base and is going to be called up more then once, you can re-display the screen much more quickly by saving the screen in memory.

RETURNS

0 is returned if no error occurred.
1 is returned if registration failed (out of memory).

RELATED FUNCTIONS

```
status = xsm_issv(screen_name);
call xsm_unsvscreen(screen_list, count);
```

t_scroll

test whether an array can scroll

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_t_scroll(field_number);
```

DESCRIPTION

This function returns 1 if the array in question is scrollable, and 0 if not. The argument `field_number` may be any field within the array.

RETURNS

1 if the array is scrolling.
0 if it is not scrolling or if no such `field_number`.

RELATED FUNCTIONS

```
status = xsm_t_shift(field_number);
```

t_shift

test whether field can shift

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_t_shift(field_number);
```

DESCRIPTION

This function returns 1 if the field in question is shiftable, and 0 if not or if there is no such field.

RETURNS

1 if field is shifting.
0 if not shifting or `field_number` is invalid.

RELATED FUNCTIONS

```
status = xsm_t_scroll(field_number);
```

tab

move the cursor to the next unprotected field

SYNOPSIS

```
call xsm_tab();
```

DESCRIPTION

If the cursor is in a field with a next–field edit and one of the fields specified by the edit is unprotected from tabbing, the cursor is moved to the first enterable position of that field. Otherwise, the cursor is advanced to the first enterable position of the next tab unprotected field on the screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

RELATED FUNCTIONS

```
call xsm_backtab();
field_number = xsm_home();
call xsm_last();
call xsm_nl();
```

tst_all_mdts

find first modified occurrence

SYNOPSIS

```
declare occurrence          fixed binary(31);
declare field_number        fixed binary(31);
field_number = xsm_tst_all_mdts(occurrence);
```

DESCRIPTION

This function tests the MDT bits of all occurrences of all fields on the current screen, and returns the base field and occurrence numbers of the first occurrence with its MDT set, if there is one. The MDT bit indicates that an occurrence has been modified, either from the keyboard or by the application program, since the screen was displayed (or since its MDT was last cleared by `xsm_bitop`).

This function returns zero if no occurrences have been modified. If one has been modified, it returns the base field number, and stores the occurrence number in `occurrence`.

RETURNS

0 if no MDT bit is set anywhere on the screen

The number of the first field on the current screen for which some occurrence has its MDT bit set. In this case, the number of the first occurrence with MDT set is returned in `occurrence`.

RELATED FUNCTIONS

```
status = xsm_bitop(field_number, action, bit);
call xsm_cl_all_mdts();
```

uinstall

install an application function

SYNOPSIS

```
declare usage              fixed binary(31);
declare func_name          char(256) varying;
declare func               entry variable;
declare language           fixed binary(31);
declare status             fixed binary(31);
status = xsm_uinstall( usage, func_name, func, language);
```

DESCRIPTION

This function installs an application routine that will be called from JAM library functions. Installation enables JAM to pass control to your code in the proper function context.

The possible values for `usage` are defined in the table below (and in the file: `smdefs.incl.pl1`). See section 2.1.1. for more detailed descriptions of the various function types.

If an application is bound with the *-retain_all* option, then JAM can find the entrypoint `func` from the name. Most functions will install themselves automatically the first time they are called. Functions may also be explicitly installed. `func_name` is the name of the function. Use the operating system subroutine `s$find_entry` to find the entry point, or use the variant `xsm_n_uinstall`, which will find it for you. `language` should be set to 1 when programming in PL/1.

Value for `usage`	*Function type*	*Section – Page*
`UINIT_FUNC`	Initialization	2.2.9. – p. 22
`URESET_FUNC`	Reset	2.2.9. – p. 22
`VPROC_FUNC`	Video processing	2.2.12. – p. 24
`CKDIGIT_FUNC`	Check digit computation	2.2.8. – p. 21
`KEYCHG_FUNC`	Keychange	2.2.4. – p. 17
`INSCRSR_FUNC`	Insert/overwrite toggle	2.2.1. – p. 11
`PLAY_FUNC`	Playback recorded keys	2.2.10. – p. 23

Value for usage	*Function type*	*Section – Page*
RECORD_FUNC	Record keys for playback	2.2.10. – p. 23
AVAIL_FUNC	Check for recorded keys	2.2 10. – p. 23
BLKDRVR_FUNC	Block Driver function	
STAT_FUNC	Status line function	2.2.11. – p. 23
DFLT_FIELD_FUNC	Default Field function	2.2.1. – p. 11
DFLT_SCREEN_FUNC	Default Screen function	2.2.2. – p. 15
DFLT_SCROLL_FUNC	Default Scroll driver	
DFLT_GROUP_FUNC	Default Group function	2.2.5. – p. 18

RETURNS

1 if function was successfully installed.
–1 if malloc failure occurred.

VARIANTS

```
status = xsm_n_uinstall( usage, func_name, language);
```

RELATED FUNCTIONS

```
call xsm_async(func, timeout);
```

ungetkey

push back a translated key on the input

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare key                fixed binary(31);
declare return_value       fixed binary(31);
return_value = xsm_ungetkey(key);
```

DESCRIPTION

This function saves the translated key given by `key` so that it will be retrieved by the next call to `xsm_getkey`. Multiple calls are permitted. The key values are pushed onto a stack (LIFO).

When `xsm_getkey` reads a key *from the keyboard*, it flushes the display first, so that the operator sees a fully updated display before typing anything. Such is not the case for keys pushed back by `xsm_ungetkey`; since the input is coming from the program, it is responsible for updating the display itself.

RETURNS

The value of its argument, or
-1 if memory for the stack is unavailable.

RELATED FUNCTIONS

```
key = xsm_getkey();
```

unsvscreen

remove screens from the save list

SYNOPSIS

```
declare screen_list        char(256) varying;
declare count              fixed binary(31);
call xsm_unsvscreen(screen_list, count);
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. This function is used to remove screens from the save list. The argument `count` specifies the number of screens to be removed from the save list. See `xsm_svscreen`.

This function can be used at any point within your code. It is not necessary for the screen to be open at the time of the call. Any memory allocated to hold the screen is freed at the time of the call unless the screen is open. The memory associated with an open screen is de-allocated when that screen is closed. If a screen is not on the save list, a call to `xsm_unsvscreen` has no effect.

RELATED FUNCTIONS

```
status = xsm_issv(screen_name);
status = xsm_svscreen(screen_list, count);
```

viewport

modify viewport size and offset

SYNOPSIS

```
declare position_row      fixed binary(31);
declare position_col      fixed binary(31);
declare size_row          fixed binary(31);
declare size_col          fixed binary(31);
declare offset_row        fixed binary(31);
declare offset_col        fixed binary(31);
call xsm_viewport(position_row, position_col, size_row,
        size_col, offset_row, offset_col);
```

DESCRIPTION

This function dynamically sizes the current screen's viewport. A viewport has a maximum size of the screen or physical display – whichever is smaller. Use `size_row` and `size_column` to specify the number of rows and columns, respectively.

You can position the viewport anywhere on the physical display. To do this, think of your physical display as a grid made up of rows and columns that are one character apart. The top left corner of your screen monitor is at position row 0, column 0 . Now use the arguments `position_row` and `position_col` to specify the coordinates of the viewport's position.

Likewise, you can also specify which row and column of the screen will initially appear at top left corner of the viewport. Again starting at row 0, column 0, count from the top left of the screen to get the coordinates for `offset_row` and `offset_col`.

This function performs range checks on all parameters and suitably modifies them if necessary. In particular, be aware that a non–positive value of `size_row` and `size_col` will set the viewport to the maximum size in that dimension.

vinit

initialize video translation tables

SYNOPSIS

```
declare video_address      bit(0);
declare status             fixed binary(31);
status = xsm_vinit(video_address);
```

DESCRIPTION

This routine is called by `xsm_initcrt` as part of the initialization process. It can also be called directly by an application program. `video_address` contains the address of a memory resident video file. Such a file must be created by the `vid2bin` and `bin2c` utilities, then compiled into the application.

RETURNS

0 if initialization is successful.
program exit if video file is invalid or if `video_address` is zero and `SMVIDEO` is undefined.

Note: The variant `xsm_n_vinit` has no return value.

VARIANTS

```
call xsm_n_vinit(video_file);
```

wcount

obtain number of currently open windows

SYNOPSIS

```
declare return_value       fixed binary(31)
return_value = xsm_wcount();
```

DESCRIPTION

This function returns the number of windows currently open. The number is equivalent to the number of windows in the window stack.

To select the screen beneath the current window, subtract 1 from the value returned by `xsm_wcount`, and then use the result as the argument to `xsm_wselect`.

This routine is useful when you are bringing another window to the top of the window stack (making the window active) with `xsm_wselect`.

RETURNS

The number of windows.
0 if the base form is the only open screen.
-1 if there is no current screen.

RELATED FUNCTIONS

```
return_value = xsm_wselect(window_number);
```

wdeselect

restore the formerly active window

SYNOPSIS

```
declare status              fixed binary(31);
status = xsm_wdeselect();
```

DESCRIPTION

This function restores a window to its original position in the window stack, after it has been moved to the top by a call to `xsm_wselect`. Information necessary to perform this task is saved during each call to `xsm_wselect`, but is not stacked. Therefore a call to this routine must follow a call to `xsm_wselect` if it is to properly restore the window to its original position. Note that `xsm_wdeselect` does not have to be called if the window ordering on the stack is acceptable.

RETURNS

-1 if there is no window to restore.
0 otherwise.

RELATED FUNCTIONS

```
call xsm_sibling(should_it_be);
return_value = xsm_wcount();
return_value = xsm_wselect(window_number);
```

window

display a window at a given position

SYNOPSIS

```
declare screen_name        char(256) varying;
declare start_line         fixed binary(31);
declare start_column       fixed binary(31);
declare status             fixed binary(31);
status = xsm_r_window(screen_name, start_line, start_column);


declare screen_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_r_at_cur(screen_name);


declare screen_address     bit(0);
declare start_line         fixed binary(31);
declare start_column       fixed binary(31);
declare status             fixed binary(31);
status = xsm_d_window(screen_address, start_line,
         start_column);


declare screen_address     bit(0);
declare status             fixed binary(31);
status = xsm_d_at_cur(screen_address);


declare lib_desc           fixed binary(31);
declare screen_name        char(256) varying;
declare start_line         fixed binary(31);
declare start_column       fixed binary(31);
declare status             fixed binary(31);
status = xsm_l_window(lib_desc, screen_name, start_line,
         start_column);


declare lib_desc           fixed binary(31);
declare screen_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_l_at_cur(lib_desc, screen_name);
```

DESCRIPTION

This set of functions is primarily intended to be used by developers who are writing their own executive. To open a window while under the control of the JAM Executive, use a JAM control string or `xsm_jwindow`.

Use `xsm_d_window`, `xsm_l_window`, or `xsm_r_window` to display `screen_name` with its upper left-hand corner at the specified line and column. The line and column are counted *from zero*. If `start_line` is 1, the window is displayed starting at the *second* line of the screen.

Use `xsm_d_at_cur`, `xsm_l_at_cur`, and `xsm_r_at_cur` to display a window at the current cursor position, offset by one line to avoid hiding that line's current display.

Whatever part of the display the new window does not occupy will remain visible. However, only the topmost (active) window and its fields are accessible to keyboard entry and library routines. JAM will not allow the cursor outside the topmost window. If you wish to shuffle windows use `xsm_wselect`.

If the window will not fit on the display at the location you request, JAM will adjust its starting position. If the window would hang below the screen and you have placed its upper left-hand corner in the *top* half of the display, the window is simply moved up. If your starting position is in the *bottom* half of the screen, the lower left hand corner of the window is placed there. Similar adjustments are made in the horizontal direction.

When you use `xsm_r_window` the named screen is sought first in the memory-resident screen list, and if found there is displayed using `xsm_d_window`. It is next sought in all the open libraries, and if found is displayed using `xsm_l_window`. Next it is sought on disk in the current directory; then under the path supplied to `xsm_initcrt`; then in all the paths in the setup variable `SMPATH`. If any path exceeds 80 characters, it is skipped. If the entire search fails, this function displays an error message and returns.

You may save processing time by using `xsm_d_window` and `xsm_d_at_cur` to display screens that are memory-resident. Use `bin2c` to convert screens from disk files, which you can modify using `jxform`, to program data structures you can compile into your application. A memory-resident screen is never altered at run-time, and may therefore be made shareable on systems that provide for sharing read-only data. `xsm_r_window` and `xsm_r_at_cur` can also display memory-resident screens, if they are properly installed using `xsm_formlist`. Memory-resident screens are particularly useful in applications that have a limited number of screens, or in environments that have a slow disk (*e.g.* MS-DOS). `screen_address` is the address of the screen in memory.

You may also save processing time by using `xsm_l_window` and `xsm_l_at_cur` to display screens that are in a library. A library is a single file containing many screens (and/or JPL modules and keysets). You can assemble one from individual screen files us-

ing the utility `formlib`. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on the number of paths searched.

The library descriptor, `lib_desc`, is an integer returned by `xsm_l_open`, which you must call before trying to read any screens from a library. Note that `xsm_r_window` and `xsm_r_at_cur` also search any open libraries.

If you want to display a form use `xsm_r_form` or one of its variants. Use `xsm_close_window` to close the window.

RETURNS

0 if no error occurred during display of the screen;
–1 if the screen file's format is incorrect;
–2 if the screen cannot be found;
–3 if the system ran out of memory but the previous screen was restored;
–5 is returned if, after the screen was cleared, the system ran out of memory.
–6 is returned if the library is corrupted.

RELATED FUNCTIONS

```
status = xsm_close_window();
status = xsm_r_form(screen_name);
status = xsm_jwindow(screen_name);
```

winsize

allow end–user to interactively move and resize a window

SYNOPSIS

```
declare status              fixed binary(31);
status = xsm_winsize();
```

DESCRIPTION

Calling `xsm_winsize` has the same effect as if the end–user had just hit the VWPT (viewport) logical key. The viewport status line appears and the user can move, resize and change the offset of the screen as well as move to any sibling windows. When the end–user hits XMIT (transmit) the previous status line is restored. If you wish to resize the viewport yourself, use `xsm_viewport`.

In order for the end–user to able to move from one window to another, the windows must be siblings. Windows are defined as siblings of one another either with `xsm_sibling` or by calling up a window as a sibling with a JAM control string. See the sections on "Viewports and Positioning" and "Control Strings" in the Author's Guide for further information.

RETURNS

–1 if call fails.
0 otherwise.

RELATED FUNCTIONS

```
call xsm_sibling(should_it_be);
call xsm_viewport(position_row, position_col, size_row,
         size_col, offset_row, offset_col);
```

wrecord

write data from a data dictionary record to a structure

SYNOPSIS

```
declare structure_ptr     bit(0);
declare record_name       char(256) varying;
declare byte_count        fixed binary(31);
call xsm_wrecord(structure_ptr, record_name, byte_count);
```

DESCRIPTION

This function writes data from fields within the current screen that are part of a common data dictionary record to a PL/1 structure. If a field is not on the current screen, then the data is read from the LDB. This routine is commonly used with `xsm_rrecord`, which reads data from a structure to a data dictionary record. If you wish to write data only from the current screen, use `xsm_wrtstruct`. To write data from a group of consecutively numbered fields, use `xsm_wrt_part`. Use `xsm_getfield` to write information from an individual field to a string.

A data structure named `record` can be created from the data dictionary file `data.dic` via the `dd2struct` utility as follows:

```
dd2struct -gPL1 data.dic
```

Each structure member is a field within a data dictionary record that is of the type specified in the Screen Editor. See "Data Type" in the Author's Guide and `dd2struct` in the Utilities Guide for further information.

Once created, the declarations may be treated exactly like any other structure declarations. The argument `struct_ptr` is the address of a variable of one of the structure types generated by `dd2struct`. The argument `record_name` is the name of the data dictionary record, from which the structure was created.

The argument `byte_count` is a pointer to an integer. Upon return from `xsm_wrecord`, the value contained in the integer will be the number of bytes or characters written to the structure. It will be 0 if an error occurred.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
call xsm_rrecord(structure_ptr, record_name, byte_count);
```

wrt_part

write part of the screen to a structure

SYNOPSIS

```
declare screen_struct      bit(0);
declare first_field        fixed binary(31);
declare last_field         fixed binary(31);
call xsm_wrt_part(screen_struct, first_field, last_field);
```

DESCRIPTION

This function writes the contents of all fields between `first_field` and `last_field` to a data structure in memory. An array and its scrolling occurrences will be copied only if the *first* element falls between `first_field` and `last_field`. Group selections are not copied. This routine is commonly used with `xsm_rd_part`, which reads part of a structure into the current screen. If you wish to write the contents of all of the fields within the screen use `xsm_wrtstruct`. To write information from a data dictionary record, use `xsm_wrecord`. Use `xsm_getfield` to write information from an individual field to a string.

A data structure named `screen` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gPL1 screen.jam
```

Each member of the structure is a field of the type specified in the Screen Editor. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other structure declaration. You can ignore the members that represent fields that do not fall within the bounds of the specified fields. However, the structure definition must contain all of the fields on screen. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`.

The arguments that represent the range of fields to be copied, `first_field` and `last_field` are passed as field numbers.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
```

```
call xsm_rd_part(screen_struct, first_field, last_field);
call xsm_wrtstruct(screen_struct, byte_count);
```

wrtstruct

write data from the screen to a structure

SYNOPSIS

```
declare screen_struct      bit(0);
declare byte_count         fixed binary(31);
call xsm_wrtstruct(screen_struct, byte_count);
```

DESCRIPTION

This function writes the contents of all of the fields within the current screen to a PL/1 structure. It will not copy group selections. This routine is commonly used with `xsm_rdstruct` which reads data from a structure to all of the fields within the current screen. If you wish to write the contents of a group of consecutively numbered fields into a structure use `xsm_wrt_part`. To write information from a data dictionary record, use `xsm_wrecord`. Use `xsm_getfield` to write the contents of an individual field into a string.

A data structure named `screen` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gPL1 screen.jam
```

Each member of the structure is a field of the type specified in the Screen Editor. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other structure declaration. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`. If you specify the type `omit`, data will not be written into the field.

The argument `byte_count` is an integer variable. `xsm_wrtstruct` will store there the number of bytes copied to the structure.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
call xsm_rd_struct(screen_struct, byte_count);
call xsm_wrt_part(screen_struct, first_field, last_field);
```

wselect

activate a window

SYNOPSIS

```
declare window_number      fixed binary(31);
declare return_value       fixed binary(31);
return_value = xsm_wselect(window_number);
```

DESCRIPTION

Although JAM allows you to display multiple windows at one time, only one window may be active. Windows may overlap each other, or may be *tiled* (no overlap). The window at the top of the window stack is the active window, and the only window accessible to library routines and keyboard entry. Use `xsm_wselect` to bring a window to the active position on top of the window stack. If any of the referenced window is hidden by an overlying window, it will be brought to the forefront of the display. In either case, the cursor is placed within the window. JAM will restore the cursor to its position when the screen was most recently de-activated.

The window to be activated is referenced by its number in the window stack. Windows are numbered sequentially, starting from the bottom of the stack. The form underlying all the windows (the base form) is window `0`, the first window displayed is `1` and so forth. Since a screen's number depends on its position on the window stack, calling `xsm_wselect` will alter a window's number as well as it position on the stack.

Alternatively, windows may be referenced by their screen name with the variant `xsm_n_wselect`. If you use this routine, you do not have to worry about keeping track of the non-active window's position on the stack. However, `xsm_n_wselect` will not find windows displayed with `xsm_d_window` or related functions, because they do not record the screen name.

Here are two different ways of using window selection. One way to use this is to select a hidden screen, update it (using `xsm_putfield`) and deselect it (using `xsm_wdeselect`). The portion of the hidden screen that is visible will be updated with the new data. Because of *delayed write* the update will be done when the next keyboard input is sought. The other method is to select a hidden screen and open the keyboard; in this case, the selected screen becomes visible, and may hide part or all of the screen that was previously active. In this way you can implement multi-page forms, or switch among several windows that *tile* the screen (do not overlap).

RETURNS

The number of the window that was made active (either the number passed, or the maximum if that was out of range).
-1 if the window was not found or the window was not open.

VARIANTS

```
return_value = xsm_n_wselect(window_name);
```

RELATED FUNCTIONS

```
call xsm_sibling(should_it_be);
return_value = xsm_wcount();
status = xsm_wdeselect();
```

Chapter 13.
Library Function Index

This chapter lists all JAM library functions, sorted by name. Function names appear on the left, and the section of the Function Reference Chapter in which the function is described appears on the right.

```
xsm_cl_all_mdts(); ..... cl_all_mdts
xsm_cl_unprot(); ..... cl_unprot
status = xsm_clear_array(field_number); ..... clear_array
status = xsm_close_window()(); ..... close_window
status = xsm_d_at_cur(screen_address); ..... window
status = xsm_d_form(screen_address); ..... form
status = xsm_d_keyset(address, scope); ..... keyset
xsm_d_msg_line(message, display_attribute); ..... d_msg_line
status = xsm_d_window(screen_address, start_line, start_column); . window
value = xsm_dblval(field_number); ..... dblval
xsm_dd_able(flag); ..... dd_able
status = xsm_deselect(group_name, group_occurrence); ..... deselect
status = xsm_dicname(dic_name); ..... dicname
offset = xsm_disp_off()(); ..... disp_off
data_length = xsm_dlength(field_number); ..... dlength
xsm_do_region(line, column, length, display_attribute, text); . do_region
status = xsm_dtofield(field_number, value, format); ..... dtofield
xsm_e_...(field_name, element, ...); ..... e_
status = xsm_e_lprotect(field_name, element, mask); ..... protect
status = xsm_e_lunprotect(field_name, element, mask); .... ..... protect
status =  xsm_e_amt_format(field_name, element, buffer); ..... amt_format
status = xsm_e_bitop(array_name, element, action, bit); ..... bitop
status = xsm_e_chg_attr(field_name, element, display_attribute); chg_attr
value = xsm_e_dblval(field_name, element); ..... dblval
data_length = xsm_e_dlength(field_name, element); ..... dlength
status = xsm_e_dtofield(field_name, element, value, format); ... dtofield
value = xsm_e_finquire(field_name, element, which); ..... finquire
field_number = xsm_e_fldno(field_name, element); ..... fldno
buffer = xsm_e_fptr(field_name, element); ..... fptr
buffer = xsm_e_ftog(field_name, element, group_occurrence); ..... ftog
status = xsm_e_fval(array_name, element); ..... fval
length = xsm_e_getfield(buffer, name, element); ..... getfield
status = xsm_e_gofield(field_name, element); ..... gofield
value = xsm_e_intval(field_name, element); ..... intval
status = xsm_e_is_no(field_name, element); ..... is_no
status = xsm_e_is_yes(field_name, element); ..... is_yes
status = xsm_e_itofield(field_name, element, value); ..... itofield
value = xsm_e_lngval(field_name, element); ..... lngval
status = xsm_e_ltofield(field_name, element, value); ..... ltofield
status = xsm_e_novalbit(field_name, element); ..... novalbit
status = xsm_e_null(field_name, element); ..... null
status = xsm_e_off_gofield(field_name, element, offset); .... off_gofield
status = xsm_e_protect(field_name, element); ..... protect
status = xsm_e_putfield(name, element, data); ..... putfield
```

```
status = xsm_e_unprotect(field_name, element); ..... protect
buffer = xsm_edit_ptr(field_number, edit_type); ..... edit_ptr
xsm_emsg(message); ..... emsg
xsm_err_reset(message); ..... err_reset
buffer = xsm_fi_path(file_name); ..... fi_path
value = xsm_finquire(field_number, which); ..... finquire
xsm_flush(); ..... flush
status = xsm_formlist(name, address); ..... formlist
buffer = xsm_fptr(field_number); ..... fptr
buffer = xsm_ftog(field_number, group_occurrence); ..... ftog
type = xsm_ftype(field_number, precision_ptr); ..... ftype
status = xsm_fval(field_number); ..... fval
field_number = xsm_getcurno()(); ..... getcurno
length = xsm_getfield(buffer, field_number); ..... getfield
buffer = xsm_getjctrl(key, default); ..... getjctrl
key = xsm_getkey()(); ..... getkey
status = xsm_gofield(field_number); ..... gofield
value = xsm_gp_inquire(group_name, which); ..... gp_inquire
length = xsm_gwrap(buffer, field_number, buffer_length); ..... gwrap
status = xsm_hlp_by_name(help_screen); ..... hlp_by_name
field_number = xsm_home()(); ..... home
xsm_i_...(field_name, occurrence, ...); ..... i_
status = xsm_i_achg(field_name, occurrence, display_attribute); .... achg
status = xsm_i_amt_format(field_name, occurrence, buffer); ... amt_format
status = xsm_i_bitop(array_name, occurrence, action, bit); ..... bitop
value = xsm_i_dblval(field_name, occurrence); ..... dblval
data_length = xsm_i_dlength(field_name, occurrence); ..... dlength
return_value = xsm_i_doccur(field_name, occurrence, count); ..... doccur
status = xsm_i_dtofield(field_name, occurrence, value, format); dtofield
value = xsm_i_finquire(field_name, occurrence, which); ..... finquire
field_number = xsm_i_fldno(field_name, occurrence); ..... fldno
buffer = xsm_i_fptr(field_name, occurrence); ..... fptr
buffer = xsm_i_ftog(field_name, occurrence, group_occurrence); ..... ftoq
status = xsm_i_fval(field_name, occurrence); ..... fval
length = xsm_i_getfield(buffer, name, occurrence); ..... getfield
status = xsm_i_gofield(field_name, occurrence); ..... gofield
field_number = xsm_i_gtof(group_name, group_occurrence, occurrence); gtof
value = xsm_i_intval(field_name, occurrence); ..... intval
lines_inserted = xsm_i_ioccur(field_name, occurrence, count); .... ioccur
status = xsm_i_is_no(field_name, occurrence); ..... is_no
status = xsm_i_is_yes(field_name, occurrence); ..... is_yes
status = xsm_i_itofield(field_name, occurrence, value); ..... itofield
value = xsm_i_lngval(field_name, occurrence); ..... lngval
status = xsm_i_ltofield(field_name, occurrence, value); ..... ltofield
```

```
status = xsm_i_novalbit(field_name, occurrence); ..... novalbit
status = xsm_i_null(field_name, occurrence); ..... null
status = xsm_i_off_gofield(field_name, occurrence, offset); . off_gofield
status = xsm_i_putfield(name, occurrence, data); ..... putfield
status = xsm_ininames(name_list); ..... ininames
xsm_initcrt(path); ..... initcrt
key = xsm_input(initial_mode); ..... input
value = xsm_inquire(which); ..... inquire
value = xsm_intval(field_number); ..... intval
status = xsm_is_no(field_number); ..... is_no
status = xsm_is_yes(field_number); ..... is_yes
old_flag = xsm_isabort(flag); ..... isabort
value = xsm_iset(which, newval); ..... iset
status = xsm_isselected(group_name, group_occurrence); ..... isselected
status = xsm_issv(screen_name); ..... issv
status = xsm_itofield(field_number, value); ..... itofield
status = xsm_jclose()(); ..... jclose
status = xsm_jform(screen_name); ..... jform
xsm_jinitcrt(path); ..... initcrt
return_value = xsm_jplcall(jplcall_text); ..... jplcall
status = xsm_jplload(module_name_list); ..... jplload
status = xsm_jplpublic(module_name_list); ..... .... jplpublic
status = xsm_jplunload(module_name); ..... jplunload
xsm_jresetcrt(); ..... resetcrt
status = xsm_jtop(screen_name); ..... jtop
status = xsm_jwindow(screen_name); ..... jwindow
xsm_jxinitcrt(path); ..... initcrt
xsm_jxresetcrt(); ..... resetcrt
old_flag = xsm_keyfilter(flag); ..... keyfilter
status = xsm_keyhit(interval); ..... keyhit
status = xsm_keyinit(key_address); ..... keyinit
buffer = xsm_keylabel(key); ..... keylabel
oldval = xsm_keyoption(key, mode, newval); ..... keyoption
scope = xsm_kscscope()(); ..... kscscope
status = xsm_ksinq(scope, number_keys, number_rows, current_row,
          maximum_len, keyset_name); ..... ksinq
xsm_ksoff(); ..... ksoff
xsm_kson(); ..... kson
status = xsm_l_at_cur(lib_desc, screen_name); ..... window
status = xsm_l_close(lib_desc); ..... l_close
status = xsm_l_form(lib_desc, screen_name); ..... form
lib_desc = xsm_l_open(lib_name); ..... l_open
status = xsm_l_window(lib_desc, screen_name, start_line,
           start_column); ..... window
```

```
xsm_last();  ..... last
status = xsm_lclear(scope);  ..... lclear
xsm_ldb_init();  ..... ldb_init
xsm_leave();  ..... leave
field_length = xsm_length(field_number);  ..... length
value = xsm_lngval(field_number);  ..... lngval
status = xsm_lreset(file_name, scope);  ..... lreset
status = xsm_lstore()();  ..... lstore
status = xsm_ltofield(field_number, value);  ..... ltofield
xsm_m_flush();  ..... flush
maximum = xsm_max_occur(field_number);  ..... max_occur
old_mode = xsm_mnutogl(screen_mode);  ..... mnutogl
xsm_msg(column, disp_length, text);  ..... msg
buffer = xsm_msg_get(number);  ..... msg_get
buffer = xsm_msgfind(number);  ..... msgfind
status = xsm_msgread(code, class, mode, arg);  ..... msgread
status = xsm_mwindow(text, line, column);  ..... mwindow
xsm_n_...(field_name, ...);  ..... n_
status = xsm_n_lclear_array(field_name);  ..... clear_array
status = xsm_n_lprotect(field_name, mask);  ..... protect
status = xsm_n_lunprotect(field_name, mask);  ..... protect
status = xsm_n_amt_format(field_name, buffer);  ..... amt_format
status = xsm_n_aprotect(field_name, mask);  ..... protect
status = xsm_n_ascroll(field_name, occurrence);  ..... ascroll
status = xsm_n_aunprotect(field_name, mask);  ..... protect
status = xsm_n_bitop(name, action, bit);  ..... bitop
status = xsm_n_chg_attr(field_name, display_attribute);  ..... chg_attr
status = xsm_n_clear_array(field_name);  ..... clear_
value = xsm_n_dblval(field_name);  ..... dblval
data_length = xsm_n_dlength(field_name);  ..... dlength
status = xsm_n_dtofield(field_name, value, format);  ..... dtofield
buffer = xsm_n_edit_ptr(field_name, edit_type);  ..... edit_ptr
value = xsm_n_finquire(field_name, which);  ..... finquire
field_number = xsm_n_fldno(field_name);  ..... fldno
buffer = xsm_n_fptr(field_name);  ..... fptr
buffer = xsm_n_ftog(field_name, group_occurrence);  ..... ftog
type = xsm_n_ftype(field_number, precision_ptr);  ..... ftype
status = xsm_n_fval(field_name);  ..... fval
length = xsm_n_getfield(buffer, name);  ..... getfield
status = xsm_n_gofield(field_name);  ..... gofield
status = xsm_n_gval(group_name);  ..... gval
value = xsm_n_intval(field_name);  ..... intval
status = xsm_n_is_no(field_name);  ..... is_no
status = xsm_n_is_yes(field_name);  ..... is_yes
```

```
status = xsm_n_itofield(field_name, value); ..... itofield
status = xsm_n_keyinit(key_file); ..... keyinit
field_length = xsm_n_length(field_name); ..... length
value = xsm_n_lngval(field_name); ..... lngval
status = xsm_n_ltofield(field_name, value); ..... ltofield
maximum = xsm_n_max_occur(field_name); ..... max_occur
status = xsm_n_novalbit(field_name); ..... novalbit
status = xsm_n_null(field_name); ..... null
number = xsm_n_num_occurs(field_name); ..... num_occurs
status = xsm_n_off_gofield(field_name, offset); ..... off_gofield
return_value = xsm_n_oshift(field_name, offset); ..... oshift
status = xsm_n_protect(field_name); ..... protect
status = xsm_n_putfield(name, data); ..... putfield
lines = xsm_n_rscroll(field_name, req_scroll); ..... rscroll
actual_max = xsm_n_sc_max(field_name, new_max); ..... sc_max
size = xsm_n_size_of_array(field_name); ..... size_of_array
status = xsm_n_unprotect(field_name); ..... protect
xsm_n_vinit(video_file); ..... vinit
return_value = xsm_n_wselect(window_name); ..... wselect
buffer = xsm_name(field_number); ..... name
xsm_nl(); ..... nl
status = xsm_novalbit(field_number); ..... novalbit
status = xsm_null(field_number); ..... null
number = xsm_num_occurs(field_number); ..... num_occurs
xsm_o_...(field_number, occurrence, ...); ..... o_
status = xsm_o_achg(field_number, occurrence, display_attribute); .. achg
status = xsm_o_amt_format(field_number, occurrence, buffer); . amt_format
status = xsm_o_bitop(field_number, occurrence, action, bit); ..... bitop
status = xsm_o_chg_attr(field_number, element,
          display_attribute); ..... chg_attr
value = xsm_o_dblval(field_number, occurrence); ..... dblval
data_length = xsm_o_dlength(field_number, occurrence); ..... dlength
return_value = xsm_o_doccur(field_number, occurrence, count); .... doccur
status = xsm_o_dtofield(field_number, occurrence, value, format);dtofield
value = xsm_o_finquire(field_number, occurrence, which); ..... finquire
field_number = xsm_o_fldno(field_number, occurrence); ..... fldno
buffer = xsm_o_fptr(field_number, occurrence); ..... fptr
buffer = xsm_o_ftog(field_number, occurrence, group_occurrence); ... ftog
status = xsm_o_fval(field_number, occurrence); ..... fval
length = xsm_o_getfield(buffer, field_number, occurrence); ..... getfield
status = xsm_o_gofield(field_number, occurrence); ..... gofield
status = xsm_o_gwrap(buffer, field_number, occurrence,
          buffer_length); ..... gwrap
value = xsm_o_intval(field_number, occurrence); ..... intval
```

```
lines_inserted = xsm_o_ioccur(field_number, occurrence, count); .. ioccur
status = xsm_o_is_no(field_number, occurrence); ..... is_no
status = xsm_o_is_yes(field_number, occurrence); ..... is_yes
status = xsm_o_itofield(field_number, occurrence, value); ..... itofield
value = xsm_o_lngval(field_number, occurrence); ..... lngval
status = xsm_o_ltofield(field_number, occurrence, value); ..... ltofield
status = xsm_o_novalbit(field_number, occurrence); ..... novalbit
status = xsm_o_null(field_number, occurrence); ..... null
status = xsm_o_off_gofield(field_number, occurrence, offset); off_gofield
status = xsm_o_putfield(field_number, occurrence, data); ..... putfield
status = xsm_o_pwrap(field_number, occurrence, text); ..... pwrap
occurrence = xsm_occur_no()(); ..... occurno
status = xsm_off_gofield(field_number, offset); ..... off_gofield
oldval = xsm_option(option, newval); ..... option
return_value = xsm_oshift(field_number, offset); ..... oshift
buffer = xsm_pinquire(which); ..... pinquire
status = xsm_protect(field_number); ..... protect
buffer = xsm_pset(which, newval); ..... pset
status = xsm_putfield(field_number, data); ..... putfield
status = xsm_putjctrl(key, control_string, default); ..... putjctrl
status = xsm_pwrap(field_number, text); ..... pwrap
reply = xsm_query_msg(message); ..... query_msg
xsm_qui_msg(message); ..... qui_msg
xsm_quiet_err(message); ..... quiet_err
status = xsm_r_at_cur(screen_name); ..... window
status = xsm_r_form(screen_name); ..... form
status = xsm_r_keyset(name, scope); ..... keyset
status = xsm_r_window(screen_name, start_line, start_column); .... window
xsm_rd_part(screen_struct, first_field, last_field); ..... rd_part
xsm_rd_struct(screen_struct, byte_count); ..... rd_struct
xsm_rescreen(); ..... rescreen
xsm_resetcrt(); ..... resetcrt
status = xsm_resize(rows, columns); ..... resize
xsm_return(); ..... return
xsm_rmformlist; ..... rmformlist
xsm_rrecord(structure_ptr, record_name, byte_count); ..... rrecord
lines = xsm_rscroll(field_number, req_scroll); ..... rscroll
status = xsm_s_val()(); ..... s_val
actual_max = xsm_sc_max(field_number, new_max); ..... sc_max
buffer = xsm_sdtime(format); ..... sdtime
status = xsm_select(group_name, group_occurrence); ..... select
xsm_setbkstat(message, display_attribute); ..... setbkstat
xsm_setstatus(mode); ..... setstatus
offset = xsm_sh_off()(); ..... sh_off
```

```
xsm_shrink_to_fit(); ..... shrink_to_fit
xsm_sibling(should_it_be); ..... sibling
size = xsm_size_of_array(field_number); ..... size_of_array
size = xsm_skinq(scope, row, softkey, value, display_attribute,
        label1, label2); ..... skinq
status = xsm_skmark(scope, row, softkey, mark); ..... skmark
status = xsm_skset(scope, row, softkey, value, attribute,
        label1, label2); ..... skset
status = xsm_skvinq(scope, value, occurrence, attribute,
        label1, label2); ..... skvinq
status = xsm_skvmark(scope, value, occurrence, mark); ..... skmark
status = xsm_skvset(scope, value, occurrence, newval, attribute,
        label1, label2); ..... skvset
outbuf = xsm_strip_amt_ptr(field_number, inbuf); ..... strip_amt_ptr
status = xsm_submenu_close()(); ..... submenu_close
status = xsm_svscreen(screen_list, count); ..... svscreen
status = xsm_t_bitop(array_number, action, bit); ..... bitop
status = xsm_t_scroll(field_number); ..... t_scroll
status = xsm_t_shift(field_number); ..... tshift
xsm_tab(); ..... tab
field_number = xsm_tst_all_mdts(occurrence); ..... tst_all_mdts
status = xsm_uinstall( usage, func, func_name); ..... uinstall
return_value = xsm_ungetkey(key); ..... ungetkey
status = xsm_unprotect(field_number); ..... protect
 xsm_unsvscreen(screen_list, count); ..... unsvscreen
xsm_viewport(position_row, position_col, size_row, size_col,
        offset_row, offset_col); ..... viewport
status = xsm_vinit(video_address); ..... vinit
return_value = xsm_wcount()(); ..... wcount
status = xsm_wdeselect()(); ..... wdeselect
status = xsm_winsize()(); ..... winsize
xsm_wrecord(structure_ptr, record_name, byte_count); ..... wrecord
xsm_wrt_part(screen_struct, first_field, last_field); ..... wrt_part
xsm_wrtstruct(screen_struct, byte_count); ..... wrtstruct
return_value = xsm_wselect(window_number); ..... ... wselect
```

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Addendum

for Updates to JAM Release 5.03

for PL/1

Part Number R332–00A

August 3, 1992

Note of Explanation

This addendum describes new features in release 5.03 of **JAM**. This addendum is for the PL/1 *Programmer's Guide*. There are separate addenda for Volumes 1 and 2 of the **JAM** 5.03 documentation set.

Several insertion pages (or A–pages) are included for new library routines and utilities in **JAM** 5.03. These pages should be inserted into your **JAM** *Programmer's Guide* and *Utilities Guide* at the appropriate location. For example, page A–195 should be inserted before page 195.

Note that the page numbers for the *Utilities Guide* refer to the August 1, 1991 printing of the **JAM** manual. Page numbers in the *Programmer's Guide* refer to the March 1, 1991 printing of the PL/1 *Programmer's Guide*.

PL/1 Programmer's Guide

Page 92: New Behavior and Return Codes for `xsm_ascroll`

The library routine `xsm_ascroll` takes as arguments a field number and an occurrence. It scrolls an array such that the requested occurrence is in the specified field. If the requested occurrence cannot be placed in the specified field because it is one of the first or last occurrences in a non–circular array, then `xsm_ascroll` scrolls the occurrence onto the screen and returns the occurrence number of the occurrence that is actually in the specified field.

Page 168: Inquiring Help Level via `xsm_inquire`

The global variable `I_INHELP` now contains the level of help that the user is in, instead of just a true/false value. There may be up to five levels of help. Use `sm_inquire` to query the value of this variable. A return of zero indicates that the user is not in help, a return of 1 through 5 indicates which help level the user is in.

Page 194: `xsm_keyoption`

Certain keys can not be translated via the `KEY_XLATE` argument to `sm_keyoption`. These are: INS, REFR, SFTS, LP, and ABORT. They may, however, be disabled via the `KEY_ROUTING` argument, or intercepted via a keychange function

Page 246: Percent Escapes in `xsm_query_msg`

Percent escapes are now supported for controlling the attributes of query messages. The sequences are the same as those for `xsm_emsg`, and detailed on page 214. Note that `%Mu` and `%Md` are not supported. Query messages from JPL can also now use percent escapes.

Page 292: MDT bits and Scrolling Arrays

When lines are inserted or deleted from scrolling arrays via INSL or DELL, the MDT bits for all occurrences after the insertion or deletion are no longer set. In a database application, this prevents the need for unnecessary processing to write potentially large amounts data that have not changed. For large arrays, it can save a significant amount of processing time.

bin2pl1

convert binary **JAM** files PL1 declare data

SYNOPSIS

`bin2pl1 [-fv]` ***PL1–file binary–file...***

OPTIONS

`-f` Overwrite an existing output file.

`-v` Generate list of files processed.

DESCRIPTION

This program converts binary files created with other **JAM** utilities into PL1 source. ***PL1–file*** is usually a new file name. (To overwrite an existing file, you must use the `-f` option.)

When the utility creates the PL1 source file, it generates a data file for each of the binary input files. The name of the data file is derived from the binary file name, with the path and extension removed, and given the extension `.incl.pl1`.

The application program should include the data file in the program that uses it. The `_d` variants of certain library routines (`d_window`, `d_form`, `d_at_cur`, `d_keyset`, `d_msg_line`) can then be used.

`bin2pl1` output files may be compiled, linked with your application, and added to the memory–resident form list. (See the **JAM** *Programmer's Guide* for more information on memory–resident lists.) The following files may be made memory–resident:

- key translation files (`key2bin`)
- setup variable files (`var2bin`)
- video configuration files (`vid2bin`)
- message files (`msg2bin`)
- JPL files (`jpl2bin`)
- screen files (`jxform`)

There is no utility to convert ***ascii–file*** back to its original binary form after using `bin2pl1`. **JAM** provides other utilities that permit two–way conversions between binary and ASCII formats. For screens, these utilities are `bin2hex` and `f2asc`.

ERRORS

`Insufficient memory available.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: Try to increase the amount of available memory.

`File "%s" already exists; use '-f' to overwrite.`
Cause: You have specified an output file that already exists.
Corrective action: Use the –f flag to overwrite the file, or use another name.

`"%s": Permission denied.`
Cause: An input file was not readable, or an output file was not writeable.
Corrective action: Check the permissions of the file in question.

copyarray

copy the contents of one array to another

SYNOPSIS

```
declare destination_fld    fixed binary(31);
declare source_fld         fixed binary(31);
declare status             fixed binary(31);
status = xsm_copyarray(destination_fld, source_fld);
```

DESCRIPTION

This routine copies the contents of the array containing `source_fld` into the array containing `destination_fld`. `source_fld` and `destination_fld` are field numbers. They may be the field number of any of element in the respective array.

The developer is responsible for insuring that the arrays are compatible. Data in source array occurrences that are too long for the destination array are truncated without warning. Data in source array occurrences that are shorter than the destination array's field length are blank filled (with respect for justification).

If the source array has more occurrences than the destination array, the data in the extra occurrences are discarded. If the source array has fewer occurrences than the destination array, trailing occurrences in the destination array are cleared of data (but not de-allocated).

`copyarray` sets the MDT bit and clears the VALIDED bit for each destination array occurrence, indicating that the occurrence has been modified and requires validation.

The variant, `xsm_n_copyarray`, searches the LDB for either array if the named field is not found on the screen. However, if the destination LDB item has a scope of 1, meaning that it is a constant, then it is not altered and the function returns –1.

RETURNS

–1 if either field is not found or if the destination array in the LDB has a scope of 1.
0 otherwise.

VARIANTS

```
status = xsm_n_copyarray(destination_name, source_name);
```

RELATED FUNCTIONS

```
status = xsm_clear_array(field_number);
length = xsm_getfield(buffer, field_number);
status = xsm_putfield(field_number, data);
```

next_sync

find next synchronized array

SYNOPSIS

```
declare field_number       fixed binary(31);
declare next_array         fixed binary(31);
next_array = xsm_next_sync(field_number);
```

DESCRIPTION

Given a field number, this function finds the next array synchronized with the given field, and returns the field number of the corresponding element in that array. The next synchronized array is defined as the one to the right. If `field_number` is in the rightmost synchronized array, the function returns the corresponding element in the leftmost synchronized array (ie– it wraps around the screen).

RETURNS

The field number of the corresponding element in the next synchronized array if there is one.
Otherwise, the field number the function was passed.

soption

set a string option

SYNOPSIS

```
declare option              fixed binary(31);
declare newval              char(256) varying;
declare oldval              char(256) varying;
oldval = xsm_soption(option, newval);
```

DESCRIPTION

Use `xsm_soption` to alter during run-time the default string options defined in `smsetup.incl.pl1`. The following table lists the valid mnemonics for `option`:

Mnemonic	*Description*
SO_EDITOR	Editor to use in JPL windows.
SO_FEXTENSION	Screen file extension.
SO_LPRINT	Operating system print command.
SO_PATH	Search path for screens and JPL procedures.

These variables are fully documented in the **JAM** *Configuration Guide*, under "System Environment and Setup Files."

RETURNS

The old value for the specified option.
0 if the option is invalid or a malloc error occurred.

RELATED FUNCTIONS

```
oldval = xsm_option(option, newval);
```

wrotate

rotate the display of sibling windows

SYNOPSIS

```
declare step                fixed binary(31);
declare status              fixed binary(31);
status = xsm_wrotate(step);
```

DESCRIPTION

If two or more sibling windows are on the top of the display, this function may be used to rotate the sequence of the sibling windows. `step` is a positive or negative integer equalling the number of screen rotations. If `step` is positive, the routine takes the top-most sibling window and makes it the last sibling window for each instance of `step`. If `step` is negative, the routine takes the last sibling window and makes it first. If `step` is zero, no rotations are performed. See the figures below.

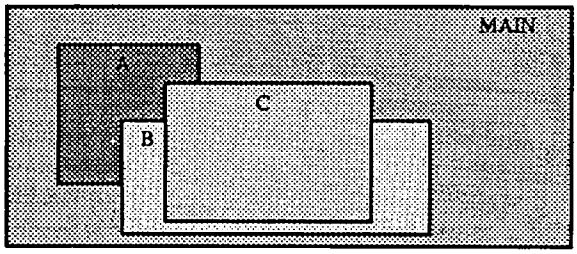


Figure 1: Screens `a`, `b`.and `c` are all siblings.Screen `main` is not a sibling.

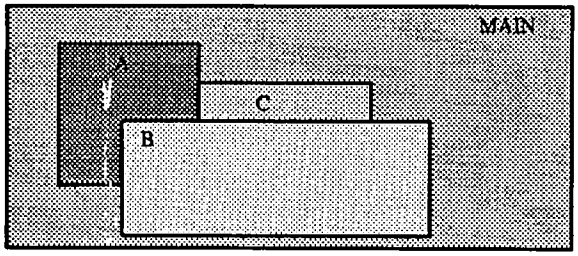


Figure 2: Executing `sm_wrotate (1)` rotates the top sibling to the bottom of the sibling stack. It rotates screen `c` behind the other two sibling windows, leaving screen `b` on top. Screen `main` is not affected.

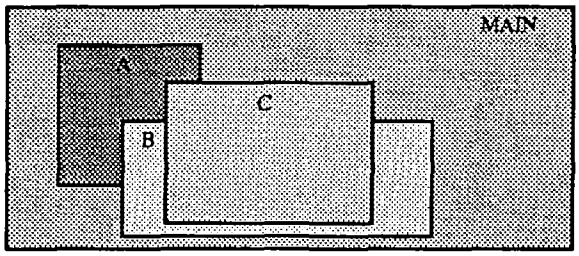


Figure 3: Executing `sm_wrotate (-1)` rotates the last sibling window to the top, putting screen `c` on top. The display is the same as Figure 1.

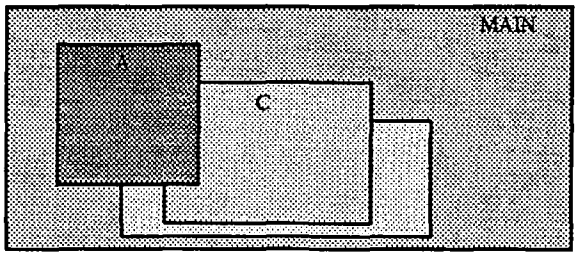


Figure 4: Executing `sm_wrotate (2)` rotates the first two sibling windows off the the top. First it rotates screen `c` to the back, then screen `b`, leaving screen `a` on top.

RETURNS

One less than the number of sibling windows on top of the window stack.
0 if there are no sibling windows

RELATED FUNCTIONS

```
call xsm_sibling(should_it_be);
```

JAM

COBOL Programmer's Guide for Stratus

This is the COBOL Programmer's manual for **JAM** Release 5. It is as accurate as possible at this time; however, both this manual and **JAM** itself are subject to revision.

Please send suggestions and comments regarding this document to:

Technical Publications Manager
JYACC, Inc.
116 John Street
New York, NY 10038

(212) 267–7722

A Note To Language Interface Users

JYACC makes every effort possible to design language interfaces that duplicate the original C Programmers Library. However, due to differences among various programming languages, an exact one to one correspondence is not always possible. In some cases, routines contained in the C version have been replaced with other routines designed to take advantage of a particular programming language's features.

Please note that your interface contains intentionally undocumented routines. Some of these routines are no longer part of **JAM**, having been replaced by more efficient routines, and are included only for backward compatibility with applications created with earlier versions of **JAM**. The rest are internal routines and are not intended to be directly accessed by developers.

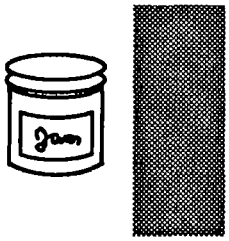
A Note To Non–UNIX Users

Throughout the manual, a forward slash (/) has been used to indicate a subdirectory. For example,

```
/usr/local/file
```

means that `file` is a file in the directory `local` which is in turn a sub–directory of `usr`, which is not the root directory.

TABLE OF CONTENTS



Chapter 1.
Introduction

This document is intended for JAM Programmers. We discuss the development and creation of executable JAM programs incorporating the Screen Manager, developer–written hook functions, and the JAM Executive. We will briefly touch on how custom executives may be written. Finally, there is a comprehensive reference of JAM library functions.

Discussions on the creation of JAM screens, data dictionaries, and keysets are found in the Author's Guide. JPL is fully documented in the JPL Programmer's Guide.

This document assumes that the reader has previously read the JAM Development Overview and the Author's Guide. The Development Overview is particularly important as the major architectural components of JAM are explained there in detail.

JAM is written in C, and the C programming interface and libraries are distributed with every license. This COBOL language interface document is an adaptation of the JAM C Programmer's Guide.

You will need to program in COBOL (or some other supported third–generation language) to accomplish the following tasks:

- To customize JAM to your environment or application by modifying the main program provided in source form with the product.
- To write hook functions that do application–specific and back–end processing during the execution of the application.
- To take full control of the application by writing an application–specific executive[1].
- To create executable JAM Programs.

As discussed in detail in the Development Overview, JAM Applications consist of screens, a data dictionary, hook functions, and an executable program. The creation of

1. It is strongly recommended that the JAM Executive be used in all but the most unusual of circumstances. A comparison of the JAM Executive with your own executive is presented in the Development Overview.

screens and data dictionaries is discussed in the Author's Guide. JPL programming is discussed in the JPL Programmer's Guide. In this chapter, we discuss how to create a JAM program. Compilation and linking are specific to platforms and operating systems and are discussed in the Installation Guide.

Two different versions of an application can be created with JAM. The Application Executable is the program delivered to the end–user to control the run time application. The JAM Authoring Executable is used to create application components and test the application during development. Only the JAM Authoring Executable will grant user access to the Screen Editor, the Data Dictionary Editor, and the Keyset Editor. The JAM Authoring Executable can only be used for the testing of applications that use the JAM Executive.

The JAM product is distributed with a plain version of the JAM Authoring Executable; one without any application–specific hook functions or records linked in. It is called `jxform`. Its use is detailed in the Author's Guide. New versions of the Authoring Executable with application–specific hook functions linked in may be created, but JAM licenses specifically forbid their distribution as runtime applications.

1.1. APPLICATION EXECUTABLE

Application Executable programs fall into two categories: those that use the JAM Executive to manage the flow of control from screen to screen, and those that use an application–specific executive. We discuss both of these approaches in the sections that follow.

1.1.1. Applications Using the JAM Executive

In applications that use the JAM Executive, most of the control flow is encapsulated in the screens. The majority of the COBOL programming task is to write hook functions (section 2. page 7) that are called by the Screen Manager or by the JAM Executive when certain events occur.

Applications that use the JAM Executive will need to be linked with the COBOL interface library `xif`, the Screen Manager library `sm`, the JAM Executive library `jm`, and, in general, the standard math library on your system.

NOTE: Refer to the Stratus Software Release Bulletin for specifics of the VOS library setup.

JYACC provides the main routine source code for applications that use the JAM Executive in a file called `jmain.cobol`. This routine performs various necessary initializa-

tions before calling the function that starts up the JAM Executive. You may want to modify this code to change JAM's default behavior.

1.1.2.

Applications Using a Custom Executive

In rare cases, a developer may choose to write a custom executive, one that is specific to a particular application. In custom executives, no library functions specific to the JAM Executive should be used. The JAM Executive functions may only be used in applications using the JAM Executive — they are listed in section 11.15. on page 87. .

Applications that do not use the JAM Executive should be linked with the COBOL interface library `xif`, the Screen Manager library `sm`, and, in general, the standard math library. If the LDB is needed, the JAM Executive library `jm` should also be linked in, but it is important the application not call any JAM Executive routines.

The "sample" application provided with JAM is a simple example of an application using a custom executive.[2] This application brings up a screen on which the end–user can enter some account data, and then save the data and call it up again. There is a help screen, tied to one of the function keys, which is implemented as a memory–resident screen, and a hook written function that verifies the area code. The discussion below outlines the basic steps that a custom executive should perform, using `sample.cobol` as an example.

To follow this discussion, you should either print this file out, or call it up in an editor. Refer to the Stratus Software Release Bulletin for the location of `sample.cobol` and the hook function `areacode.cobol`.

Declarations

After the Environment Division, a record is copied into the File Section of the Data Division. The record was created from the "sample1" screen via the `f2struct` utility. Next, a memory–resident help screen is copied into the Working Storage Section. This file was created via the `bin2cob` utility from a binary JAM screen file. Various variable and parameter declarations follow, including the Header files mentioned below.

Header Files

JAM user defines are copied in as necessary, depending on the library routines utilized in the program. The documentation for each library routine indicates which, if any, header files are required.

2. Note that JPL is available to applications that do not use the JAM Executive. Note also that hook functions may be installed and used in applications that do not use the JAM Executive. These applications, however, will not be able to use control strings.

Screen Manager Initialization

The Screen Manager and the terminal are initialized in the Procedure Division with a call to `xsm_initcrt`. Since an empty string is passed as the argument, the search path for screens is expected to be found in the environment.

Install Hook Functions

Most Screen Manager hook functions are installed via the `-retain_all` argument to the `bind` command. This is the case for the hook function `areacode`, which is called as a field validation function. For certain types of hook functions, explicit installation is necessary and should occur here—after initialization, but before displaying the first screen. The various types of hook functions and their installation are described in detail in Chapter 2.

Display the Main Form

After initialization is complete, the screen `sample1.frm` is opened as a form with a call to `xsm_r_form`. If an error occurs, the program will terminate.

Activate Screen

`sample1.frm` is activated within a loop. The loop terminates if the user strikes the EXIT key, which causes the routine `xsm_input` to return with the return code EXIT defined in `smkeys.incl.cobol`. The actual data entry, cursor movement, help processing, character edit masking, and validation are handled within `xsm_input`, so the programmer need not be concerned with them. Whenever the user strikes TRANSMIT, EXIT, or some other function key, `xsm_input` returns control to the calling program. In this case, the PF2, PF3 and EXIT keys cause specific actions. All other function keys cause a beep and the while loop to continue, calling `xsm_input` again.

Open a Window

The PF3 key brings up the memory–resident screen that was installed earlier, and then waits for the user to press a key.

Close a Window

During the run of any application, there is always a form displayed. When a new form is displayed, all existing screens are implicitly closed. Windows, however, need to be explicitly closed if the application is to retreat to an underlying screen. After the PF3 window is displayed, when the user strikes a key the program calls `xsm_close_window` to close this window.

Handle Errors

The executive should have a facility to handle errors. The PF2 key triggers a subroutine which opens a window allowing the user to save or read data. While the specifics of this

data manipulation are beyond the scope of this introductory discussion, the second to last line of the source listing illustrates the error handling routine `xsm_err_reset`, which displays an error message on the status line. The routine takes a single string argument, and places that string on the status line. The user is forced to acknowledge the error by striking the space bar[3].

Reset the Terminal

Before the application terminates, it calls `xsm_resetcrt` to reset terminal characteristics to a state expected by the operating system.

1.2.
AUTHORING EXECUTABLE

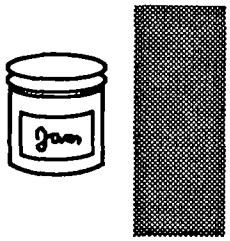
The Authoring Executable must use the **JAM** Executive, and may have developer–written hook functions linked in. The main routine for the Authoring Executable is provided in source form in a file called `jxmain.cobol`. You may want to modify that file to change the default behavior of the authoring tool `jxform`. It is strongly suggested that **JAM** developers read and understand this code, as it is instructive and may help with an understanding of the product.

The compiled Authoring Executable may be called with the optional command–line switch `-e`. This will cause the authoring tool to start up directly within the Screen Editor (as opposed to starting up in application mode).

Authoring executables must be linked with the COBOL interface library `xif`, the **JAM** Authoring Library `jx`, the **JAM** Executive library `jm`, the Screen Manager library `sm`, and, in general, the standard math library. Since these executables are linked with the **JAM** Authoring Library `jx`, they may not be re–sold or distributed on machines for which there is no software license from JYACC. This restriction applies *only* to Authoring Executables, which are intended for application *development* only.

NOTE: Refer to the Stratus Software Release Bulletin for specifics of the VOS library setup.

3. The developer may change the way messages are acknowledged with the library routine `XSM_OPTION`.



Chapter 2.
Hook Functions

The primary coding task facing JAM programmers is writing hook functions. These functions, which are called by the JAM Executive and by the Screen Manager when certain well–defined events occur, are written in COBOL[5].

In this chapter, we discuss how hook functions are written and installed. They must also be compiled and linked into the JAM Application (or Authoring) Executable: see the Installation Guide for details of that. We also discuss what JAM events have hooks accessible to developers and what arguments are passed to hook functions from any given hook. Finally, we discuss in detail the various types of hook functions, showing examples of some of them, and explaining how they are installed and used.

2.1.
PREPARATION AND INSTALLATION

Hook functions, once properly installed, are called at certain well– defined JAM events. These events are outlined below in section 2.1.1. and discussed in detail later in the chapter.

There are many events that have hooks accessible to developers. JAM passes different arguments to the various hook functions, and interprets the return codes differently for each one. It is important that hook functions process the arguments that are passed correctly, and that they return meaningful codes based on the events to which they are attached.

Hook functions are installed individually, and are called at runtime by JAM when a certain event type occurs. Most hook functions are called by the Screen Manager. However,

5. Hook functions may also be written in C and other third–generation programming languages for which JYACC supports a language interface. In particular, Fortran, Cobol and PL/1 are available for JAM on some platforms.

the hook functions invoked with control strings are called by the JAM Executive, and will only be accessible to applications using a custom executive through JPL.

2.1.1.

Types of Hook Functions

There are twenty–two installable hook function types, six of which are installed when the application is bound and sixteen of which are installed as individual functions. They are briefly outlined below, and discussed in detail later in the document:

- `FIELD-FUNC`
  These functions are installed using the `-retain_all` argument of the `bind` command. The functions on this list may be designated in the Screen Editor to be called by the Screen Manager as field entry, exit or validation functions for specific fields. The JPL `atch` verb may also be used to access these functions.

- `GROUP-FUNC`
  These functions are installed using the `-retain_all` argument of the `bind` command. These functions may be designated in the Screen Editor to be called by the Screen Manager as group entry, exit or validation functions for specific groups (Radio Buttons and Checklists).

- `SCREEN-FUNC`
  These functions are installed using the `-retain_all` argument of the `bind` command. These functions may be designated in the Screen Editor to be called by the Screen Manager as screen entry or exit functions on particular screens.

- `CONTROL-FUNC`
  These functions are installed using the `-retain_all` argument of the `bind` command. These functions may be entered and invoked from control strings. They are often associated with function keys and menus in the Screen Editor or with the `xsm_putjctrl` library call. The JPL `call` verb can invoke control functions.

- `DFLT-FIELD-FUNC`
  This is an individual function. It is installed using the library routine `xsm_n_uinstall`. Once installed, it is called on entry, exit and validation for all fields.

- `DFLT-GROUP-FUNC`
  Similar to the `DFLT-FIELD-FUNC`, this individual function is called on entry, exit, and validation for all groups.

- DFLT-SCREEN-FUNC
  Individual function called on entry and exit for all screens.

- KEYCHG-FUNC
  Individual function called whenever **JAM** reads a key from the keyboard. This allows for the application to intercept and process (and possibly translate) keystrokes at the logical key level.

- INSCRSR-FUNC
  Individual function called by **JAM** whenever the keyboard entry mode toggles between insert and overstrike mode. This allows an application to update the display, if desired, to provide an indication of the new mode. Often used if there is no ability to change cursor styles between insert and overstrike modes.

- CKDIGIT-FUNC
  Individual function called by **JAM** for check digit validation of numeric fields. Only necessary if the default check–digit algorithm provided with **JAM** is not sufficient.

- UINIT-FUNC
  Individual function called just before the Screen Manager and the physical display are initialized at the start of the application.

- URESET-FUNC
  Individual function called just after the Screen Manager and the physical display are closed and reset at the end of the application, even if the application aborts ungracefully.

- RECORD-FUNC
  Individual function used to record keystrokes so they can be played back for tutorials or for regression testing.

- PLAY-FUNC
  Individual function used to playback recorded keys.

- AVAIL-FUNC
  Individual function used in advanced record/playback algorithms.

- STAT-FUNC
  Individual function used to intercept **JAM** status line processing and alter or replace it.

- VPROC-FUNC
  Individual function used to intercept **JAM** video processing and to alter or replace it.

- BLKDRVR-FUNC
  This is an individual function that acts as a block mode terminal driver. This is discussed in section 10.1.3.

- ASYNC-FUNC
  Individual function called asynchronously when JAM is waiting for keyboard input. This is installed via the library routine xsm_async. Often used to poll external systems for mail delivery or the availability of data over a communications line.

2.1.2.
Installing Functions

As mentioned above, certain hook functions must be installed explicitly with the library routines xsm_n_uinstall or xsm_async, others are installed using the -retain_all argument of the bind command.

xsm_n_uinstall is called with three arguments. The first argument identifies the type of function being installed, and may be one of the following values:

```
UINIT-FUNC          CKDIGIT-FUNC        STAT-FUNC
URESET-FUNC         BLKDRVR-FUNC        DFLT-FIELD-FUNC
VPROC-FUNC          PLAY-FUNC           DFLT-SCREEN-FUNC
KEYCHG-FUNC         RECORD-FUNC         DFLT-GROUP-FUNC
INSCRSR-FUNC        AVAIL-FUNC
```

The second argument is the name of the function. The third argument identifies the language. This argument should be 1 for all programming languages except C.

xsm_async is used exclusively for installing asynchronous functions. It takes as arguments the address of the function and a timeout period.

The other function types, which are installed via the -retain_all argument to the bind command, are the following:

```
FIELD-FUNC
SCREEN-FUNC
CONTROL-FUNC
GROUP-FUNC
```

2.2.
WRITING HOOK FUNCTIONS

Arguments passed to hook functions and return values received from hook functions vary from hook to hook. In this section, we discuss the various JAM hooks in detail.

2.2.1.

Hook Function Return Codes

To set a return value, declare a return value argument as `PIC 9(5) comp-5` in your COBOL routine, and `move` the appropriate value to it. Then use the following syntax when exiting the routine:

```
exit program with return-value.
```

2.2.2.

Field Functions

The Screen Manager will call field functions, if specified, on field entry, field exit, and field validation. Calls to field entry and field exit functions are guaranteed to be paired for any given field.

A single default field function may also be installed. It will be invoked on entry, exit, and validation for every field. The default field function must be installed explicitly as `DFLT-FIELD-FUNC` via `xsm_n_uinstall`.

JPL procedures may be directly specified as field functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl fieldfunc`.

Field Function Invocation

Field functions are called for field entry whenever the cursor enters a field, including when the field containing the cursor is activated by virtue of an overlying window being closed. Field functions are called for field exit whenever the cursor leaves a field, including when the field is exited because a window is popped up over the existing screen. Field functions are called for validation whenever the field is validated. This occurs at the following times:

- As part of field validation, when you exit the field or scroll to the next occurrence by filling it or by hitting TAB or RETURN key. The BACK-TAB and arrow keys do not normally cause validation. Field functions are called for validation only after the field's contents pass all other validations for the field.
- As part of screen validation when the XMIT key is struck.
- When the application code calls library routines for field validation.

Field functions may also be invoked from JPL with the `atch` verb.

For fields that are members of menus, radio buttons, or checklists, the validation function is not called as part of validation. The validation function for such fields is called instead when that field is selected. For checklist fields, the field validation function is also called when the field is deselected.

Field functions specified for field entry via the Screen Editor are invoked after any installed default field function. Field functions specified for field exit or validation via the Screen Editor are called before any installed default field function.

Field Function Arguments

All field functions receive four arguments:

1. The field number as an integer.
2. A buffer containing a copy of the field's contents.
3. The occurrence number of the data as an integer.
4. An integer containing the VALIDED and MDT bits associated with the item or field, and additional flags indicating the circumstances under which the function was called.

The information in the fourth parameter includes the VALIDED and MDT bits and several flags indicating why the function was called. The following values are defined in the file `smflags.incl.cobol`:

- `VALIDED-BIT`
  If this is set, the field has passed all its validations and has not been modified since. This value is set to 32.
- `MDT-BIT`
  If this is set, the field data has been changed either from the keyboard or from the application code since the current screen was opened[7]. JAM never clears this bit. The application code may clear it directly with the `xsm_bitop` library routine. This value is set to 64.
- `K-ENTRY`
  If set, the field function was called on field entry. This value is set to 128.
- `K-EXIT`
  If set, the field function was called on field exit[8]. This value is set to 16.

7. Note that when the screen is being opened, when the screen entry function modifies data in a field the `MDT` bit is not set. However, when the screen is exposed by virtue of an overlaid window being closed, modification of field data in the screen entry function will cause the `MDT` bit to be set.

8. Note that if neither `K-ENTRY` nor `K-EXIT` are set, the field is being validated.

■K-EXPOSE

If set, the field function was called because a window overlying the screen on which the field resides was opened or closed[9]. This value is set to 256.

■K-KEYS

The following values indicate which keystroke or event caused the field to be entered, exited, or validated. The remainder in the fourth parameter to the field function after the above flags have been checked for and subtracted out, should be tested for equality against one of the six values below:

■K-NORMAL

If set, 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".

■K-BACKTAB

If set, the BACKTAB key caused the cursor to enter or exit the field in question.

■K-ARROW

If set, an arrow key caused the cursor to enter or exit the field in question.

■K-SVAL

If set, the field is being validated as part of screen validation.

■K-USER

If set, the field is being validated directly from the application with the `xsm_fval` library routine.

■K-OTHER

If set, some key other than backtab, arrow or those mentioned as "normal" caused the cursor to enter or exit the field in question.

Field functions are called for validation regardless of whether the field was previously validated. They may test the VALIDED and MDT bits to avoid redundant processing.

Field Function Return Codes

Field functions called on entry or exit should return 0. Field functions called for validation should return 0 if the field contents pass the validation criteria. Any non–zero return code should indicate that the field does not pass validation. If the returned value is 1, the cursor will not be repositioned to the offending field. Any other non–zero return value will cause the cursor to be repositioned to the field that failed the validation. This is useful

9. This means that if both K-ENTRY and K-EXPOSE are set, the field is being exposed. If K-EXIT and K-EXPOSE are set, the field is being hidden.

because when the entire screen is undergoing validation, the field that fails validation may not be the field where the cursor is.[10]

Example Field Function

The following code illustrates how to interpret the fourth argument passed to a field function, and how to set the function's return value.

```
working-storage section.

copy "smflags.incl.cobol".

01 expose-call                  pic S(9)9 comp-5.
01 exit-call                    pic S(9)9 comp-5.
01 entry-call                   pic S(9)9 comp-5.
01 mdt-is-set                   pic S(9)9 comp-5.
01 val-is-set                   pic S(9)9 comp-5.
01 why-called                   pic S(9)9 comp-5.
01 fld-amount                   pic S(9)9 comp-5.
01 auth-level                   pic S(9)9 comp-5.
01 data-buffer                  display-2 pic x(256).
01 ret-value                    pic S(9)9 comp-5 value 0.

linkage section.

01 fld-no                       pic S(9)9 comp-5.
01 fld-contents                 display-2 pic x(256).
01 occur-no                     pic S(9)9 comp-5.
01 misc-flags                   pic S(9)9 comp-5.

procedure division using fld-no, fld-contents, occur-no, misc-flags.
field-func.

* the following code shows how to break up misc-flags into its
* components.  the components are not actually tested here.

    move zero to expose-call.

    if misc-flags >= K-EXPOSE
        move 1 to expose-call
        subtract K-EXPOSE from misc-flags.

    move zero to entry-call.

    if misc-flags >= K-ENTRY
        move 1 to entry-call
        subtract K-ENTRY from misc-flags.
```

10. In many cases, it is better for the field validation function itself to reposition the cursor before displaying an error message, otherwise the error message might be misleading.

```
    move zero to mdt-is-set.

    if misc-flags >= MDT-BIT
        move 1 to mdt-is-set
        subtract MDT-BIT from misc-flags.

    move zero to val-is-set.

    if misc-flags >= VALIDED-BIT
        move 1 to val-is-set
        subtract VALIDED-BIT from misc-flags.

    move zero to exit-call.

    if misc-flags >= K-EXIT
        move 1 to exit-call
        subtract K-EXIT from misc-flags.

    move misc-flags to why-called.
* validation routine showing use of return value.
* csm_intval gives computational value.

    call "csm_intval" using fld-no giving fld-amount.

    if fld-amount < 10000
        go to done.

* if amount requires authorization, check contents of field
* named "auth_level".

    move "auth_level" to data-buffer.
    call "csm_n_intval" using data-buffer giving auth-level.

* if authorization level is too low, display error message
* and set ret-value to indicate error.

    if auth-level < 3
        move -1 to ret-value
        move "authorization required" to data-buffer
        call "csm_quiet_err" using data-buffer.

done.
    exit program with ret-value.
```

2.2.3.

Screen Functions

The Screen Manager will call screen functions, if specified, on entry and exit of screens. Calls to screen entry and screen exit functions are guaranteed to be paired for each screen.

A single default screen function may be installed. It will be invoked on entry and exit for every screen. The default screen function is installed as `DFLT-SCREEN-FUNC` via `xsm_n_uinstall`. Screen functions specified as entry or exit functions for a screen via the Screen Editor are installed via the `-retain_all` argument to the `bind` command. JPL procedures may also be directly specified as screen functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl screen-func`.

Because of the way LDB processing and form stack handling is done, it is neither recommended nor supported to call any form or window display library routines from screen entry or exit functions. If it is necessary to display windows at screen entry, the library routine `xsm_ungetkey` can be invoked, passing as the argument a function key with a control string that brings up a window.

Screen Function Invocation

Screen functions are called for screen entry whenever a screen is opened. Screen functions are called for screen exit whenever a screen is closed. Optionally, screen functions may also be called for entry when a screen is exposed by virtue of a window overlaying it being closed or deselected, and called for exit when a window is popped up or selected over the screen in question. This is not the default behavior because it would introduce incompatibilities with earlier releases of **JAM**.

If you are not concerned with compatibility with earlier releases, it is strongly suggested that you make the following library function call near the beginning of your application, enabling the calling of screen functions when screens are exposed or hidden:

```
call "xsm_option" using EXPHIDE_OPTION, ON_EXPHIDE
```

Screen functions specified for screen entry via the Screen Editor are invoked after any installed default screen function. Screen functions specified for screen exit via the Screen Editor are called before any installed default screen function.

Screen Function Arguments

All screen functions receive two arguments:

1. The screen name.
2. An integer containing contextual information about the circumstances under which the function was called.

The contextual information in the second parameter includes the following values:

■`K-ENTRY`

If this is set, the function was called on screen entry.

- K-EXIT
  If this is set, the function was called on screen exit.
- K-EXPOSE
  If this is set, the function was called because the screen was selected or deselected, or because a window was popped over the screen or a window that used to be overlaid on the screen was closed[11].
- K-NORMAL
  If set, a "normal" call to xsm_close_window caused the screen to close.
- K-OTHER
  If set, the screen is being closed because another form is being displayed or because xsm_resetcrt is called.

Screen Function Return Codes

All screen functions should return 0.

2.2.4.

Control Functions

Control functions are called by the JAM Executive in the processing of control strings and by JPL routines that call COBOL functions. The JAM Executive will call control functions, if specified and installed, when control strings that start with a caret (^) are executed. JPL procedures may also execute control functions by using the call verb.

There is no default control function. Control functions are installed via the -retain_all argument to the bind command. JPL procedures may be directly specified as control functions by preceding the name of the procedure in a control string with the string "jpl ".

A number of control functions of general use are built in to JAM. These built-ins can be used by any JAM application. They are listed in Chapter 4.

Control Function Invocation

Control functions are called by the JAM Executive when a control string starting with a caret is processed. Such control strings are often attached, via the Screen Editor, to function keys or to menu selections in control fields. In addition, the JPL verb call can be used to invoke control functions.[12]

11. If both K-ENTRY and K-EXPOSE are set, the screen is being uncovered and activated by virtue of an overlaid window being closed. If both K-EXIT and K-EXPOSE are set, the screen is being covered and deactivated by virtue of a window being popped up over it.

Control Function Arguments

Control functions receive a single argument, namely a buffer containing a copy of the control string that invoked the function, without the leading caret. It is only the first word on the control string that identifies the function, the rest of the string may contain arbitrary data that can be parsed and used as arguments.

Control Function Return Codes

Control functions may return any integer. The return value from a control function may be used for conditional control branching in target lists (see the Authoring Guide). If there is no target list, and the control string returns a function key which has an associated control string in it's own right, then that control string is executed.

2.2.5.

Key Change Functions

The key change function is called by the Screen Manager as keys are read from the keyboard from within the library routine `xsm_getkey`, which is called in the input processing for all keys by **JAM**. Only one individual keychange function may be installed at a time.

Keys placed on the queue with the library routine `xsm_ungetkey` or with the built–in control function `^jm-keys` are not processed by the installed key change function.

The key change function is installed as `KEYCHG-FUNC` via `xsm_n_uinstall`.

Key Change Function Invocation

The key change function is called exactly once for every key read in from the keyboard or supplied by the playback hook function described in section 2.2.11..

Key Change Function Arguments

The key change function is passed a single integer argument, namely the **JAM** logical key that was read from the keyboard or received from the playback hook function.

Key Change Function Return Codes

The key change function returns the key to be substituted for the one passed as an argument. Any key returned to `xsm_getkey` will be returned by `xsm_getkey` to its caller.

12. The JPL call verb does not execute control strings. It looks for functions to call.

However, if the key change function returns 0, `xsm_getkey` will get the next key from the keyboard[13].

2.2.6.

Group Functions

The Screen Manager will call group functions, if specified, on entry, exit, and validation of radio buttons and checklists. Calls to group entry and group exit functions are guaranteed to be paired for each group.

A single default group function may be installed. It will be invoked on entry, exit, and validation for every group. The default group function is installed as `DFLT-GROUP-FUNC` via `xsm_n_uinstall`. Group functions specified as entry, exit, or validation functions for a given group in the Screen Editor are installed via the `-retain_all` argument to the `bind` command. JPL procedures may also be directly specified as group functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl groupfunc`.

Please note that field validation functions for fields that are members of groups or menus are called at selection and, in the case of checklists, deselection as discussed above in section 2.2.2. on page 11.

Group Function Invocation

Group functions are called for group entry whenever the cursor enters a group, including the times when the group containing the cursor is activated by virtue of an overlying window being closed. Group functions are called for group exit whenever the cursor leaves a group, including the times when the group is left because a window is popped up over the existing screen. Group functions are called for validation whenever the group is validated. This occurs at any of the following times:

- As part of group validation, when you exit the group by hitting TAB or making a selection from an autotab group. The BACKTAB and arrow keys do not normally cause validation.
- As part of screen validation when the XMIT key is struck.
- When the application code calls library routines for group validation.

Group functions specified for group entry via the Screen Editor are invoked after any installed default group function. Group functions specified for group exit or validation via the Screen Editor are called before any installed default group function.

13. See the library routine `XSM_KEYOPTION` for a different method of changing the function of a logical key.

Group Function Arguments

All group functions receive two arguments:

1. The group name.
2. An integer containing contextual information about the validation state of the group and the circumstances under which the function was called.

The information contained in the third argument to group functions is identical to that passed in the fourth argument to field functions. See section 2.2.2. on page 12 for an explanation.

Group functions are called for validation regardless of whether the group was previously validated. They may test the VALIDED and MDT bits to avoid redundant processing.

Group Function Return Codes

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. Any non–zero return code should indicate that the group does not pass validation. If the returned value is 1, the cursor will not be repositioned to the offending group. Any other non–zero return value will cause the cursor to be repositioned to the group that failed the validation.

2.2.7.

Asynchronous Functions

The installed asynchronous function is called periodically by the Screen Manager while the keyboard input routine waits for user input. It can be used to poll or otherwise manipulate communications resources, or to update the display on the screen.

The asynchronous function is installed individually as ASYNC-FUNC via the library routine `xsm_async`.

Asynchronous Function Invocation

The asynchronous function is called from the very lowest level of JAM keyboard input. When the asynchronous function is installed, the device driver clock on the terminal input device is set to time out on its character read operation, and if a character is not read in that time interval the asynchronous function is invoked before another character read operation is attempted. The time out interval is specified when the function is installed. The time out is measured in tenths of seconds. The maximum interval is 255 (25.5 seconds).

Asynchronous Function Arguments

The asynchronous function is passed no arguments.

Asynchronous Function Return Codes

The asynchronous function should generally return 0. If it returns –1, it will not be called again until at least one additional character has been read from the keyboard. The asynchronous function may return a key, which will be returned to `xsm_getkey` and on to the application. If that key is a JAM logical key, no further translation will be done. If the asynchronous function returns a data character, JAM will interpret it as a physical keyboard stroke.

2.2.8.

Insert Toggle Functions

The Screen Manager will call the Insert Toggle Function when switching between input and overstrike mode for data entry. Generally this hook function will be used to update some aspect of the display informing the user of the current mode.

The insert toggle function is installed individually as `INSCRSR-FUNC` via `xsm_n_uinstall`. JAM automatically installs an insert toggle function that changes the cursor style when the mode is changed. If an application installs its own insert toggle function, the JAM function will be de–installed, and the new insert toggle function may want to call the function directly.

Insert Toggle Function Invocation

The function will be invoked by JAM whenever the data entry mode shifts from insert to overstrike mode or from overstrike to insert mode. Most often, this occurs when the end–user strikes the INSERT key.

Insert Toggle Function Arguments

One integer argument is passed to the insert toggle function. It specifies the mode. If its value is 1, JAM is entering insert mode. If it is 0, JAM is entering overstrike mode.

Insert Toggle Function Return Codes

The insert toggle function should return 0.

2.2.9.

Check Digit Functions

The Screen Manager will call the check digit function for any field that is marked for check digit in the Screen Editor. It may be used to implement any desired check–digit algorithm. If there is no check digit function installed in the application, JAM will use the default library function `xsm_ckdigit`. A new check digit function is installed as `CKDIGIT-FUNC` via the library routine `xsm _n_uinstall`.

Check Digit Function Invocation

The check digit function is called by JAM during validation of fields marked for check digit.

Check Digit Function Arguments

The check digit function is passed the following arguments:

1. The integer number of the field undergoing validation.
2. The field contents.
3. The integer occurrence number for the data undergoing validation.
4. The integer modulus as specified in the Screen Editor.
5. The integer minimum number of digits as specified in the Screen Editor.

Check Digit Function Return Codes

The check digit function should return 0 if the field passes the check digit validation. If a non–zero value is returned, the cursor is positioned to the offending field and the field is not marked as validated. It is assumed that the check digit function display its own error messages.

2.2.10.

Initialization and Reset Functions

The initialization and reset functions are called by the Screen Manager on display setup and display reset respectively. The initialization function can be used to set the terminal type and the reset function can be used to handle any cleanup that the application needs to do whether it is terminated gracefully or not.

Initialization and reset functions are installed individually as UINIT-FUNC and URESET-FUNC respectively via calls to xsm_n_uinstall.

Initialization and Reset Function Invocation

The initialization function is called from the library routine xsm_initcrt. When it is called, JAM has not yet allocated its required memory structures, and the physical display characteristics are still untouched by JAM. In general, it is suggested that hook functions be installed after initialization with xsm_initcrt, but clearly this is an exception. The initialization function must be installed before xsm_initcrt is called. This function is installed as UINIT-FUNC via the library routine xsm_n_uinstall.

The reset function is called from the library routine xsm_resetcrt after JAM has released its memory and reset the physical display characteristics. Since the JAM abort routine xsm_cancel calls xsm_resetcrt before the application terminates, the reset function is generally called at application exit whether the exit is graceful or not[14]. This function is installed as URESET-FUNC via the library routine xsm_n_uinstall.

Initialization and Reset Function Arguments

The initialization function is passed a single argument, namely a 30 byte character buffer into which it may place the null–terminated string mnemonic identifying the terminal type in use. This is primarily of use on operating systems without an environment. This function can be used to obtain the terminal type in some system–specific way.

The reset function is passed no arguments.

Initialization and Reset Function Return Codes

Both the initialization and reset hook functions should return 0.

2.2.11.

Recording and Playing Back Keystrokes

The Screen Manager provides hooks for recording and playing back keystrokes. This facility could be used to implement simple macro capabilities, or to perform regression testing on a JAM application. The developer should ensure that the record and playback functions are not in use simultaneously.

14. Interrupt handlers may need to be set by the developer to insure that XSM_CANCEL is called at all the necessary hardware and software interrupt signals. It is suggested that this setup be done in the function installed as an initialization function.

Record and playback functions are installed individually as RECORD-FUNC and PLAY-FUNC respectively via xsm_n_uinstall.

Record/Playback Function Invocation

The record function is called from xsm_getkey when it has a translated key value in hand that it is about to return to the application. The playback function is called from xsm_getkey, when installed, in place of a read from the keyboard[15]. For accurate regression testing, the playback function may need to pause and flush the output to simulate a realistic rate of typing, and may need to call the asynchronous function, if there is one.

Record/Playback Function Arguments

The record function is passed a single integer, which is the JAM logical key to record. Generally that key is recorded in some fashion for a possible playback at a later date. The playback function receives no arguments.

Record/Playback Function Return Codes

The record function should return 0. The playback function should return the logical key that was recorded at an earlier time.

2.2.12.

Status Line Functions

The status line function is called by the Screen Manager whenever the status line is about to be flushed, or physically written to the terminal device. It is intended for use on terminals that require unusual status line processing, beyond the scope of the generic code, but other uses are possible.

The status line function is installed individually as STAT-FUNC via xsm_n_uinstall.

Status Line Function Invocation

The status line function is called when the status line is about to be physically written to the terminal display. Because of delayed write, this may or may not be at the time when the functions that specify message line text are actually called.

15. Since characters are recorded after processing by the key change function but played back before key change translation, some key change functions may interfere with the accurate playback of recorded keystrokes. See the description of XSM_GETKEY in the Programmer's Reference Manual for more information.

Status Line Function Arguments

The status line function receives no arguments. It can access copies of the text and attributes about to be flushed to the status line using the following library routine calls:

```
call "xsm_pinquire" USING SP-STATLINE GIVING STAT-TEXT.
call "xsm_pinquire" USING SP-STATATTR GIVING STAT-ATTR.
```

Status Line Function Return Codes

If the status line function returns 0, JAM continues its usual processing and actually writes out the status line. If the function returns any other value, JAM assumes that the physical write of the status line was handled in the hook function.

2.2.13.

Video Processing Functions

The Screen Manager calls the developer-installed video processing function to allow for special handling of various video sequences by the application. This is a specialized hook required only when the JAM video file is unable to provide support for a particular type of terminal.

The video processing function is installed individually as `VPROC-FUNC` via `xsm_n_uinstall`.

Video Processing Function Invocation

The video processing function is called by JAM's output routine just before a video output operation is about to begin.

Video Processing Function Arguments

The video processing function receives two arguments. The first is an integer video processing code defined in the header file `smvideo.incl.cobol` and outlined in the table below. The second is an array of integers with parameters for the video processing code. The number of parameters passed depends on the operation as shown in the table below. For video processing codes that require no arguments, a NULL is passed.

Code	*Operation Description*	*# of params*
V-ARGR	remove area attribute	
V-ASGR	set area graphics rendition	11

Code	*Operation Description*	*# of params*
V-BELL	visible alarm sequence	
V-CMSG	close message line	
V-COF	turn cursor off	
V-CON	turn cursor on	
V-CUB	cursor back (left)	1
V-CUD	cursor down	1
V-CUF	cursor forward (right)	1
V-CUP	set cursor position (absolute)	2
V-CUU	cursor up	1
V-ED	erase entire display	
V-EL	erase to end of line	
V-EW	erase window to background	5
V-INIT	initialization string	
V-INSON	set insert cursor style	
V-INSOFF	set overstrike cursor style	
V-KSET	write to soft key label	2
V-MODE4	single character graphics mode (also V-MODE5, 6)	
V-MODE0	set graphics mode (also V-MODE1, 2, 3)	
V-OMSG	open message line	
V-RESET	reset string	
V-RCP	restore cursor position	
V-REPT	repeat character sequence	2

Code	*Operation Description*	*# of params*
V-SCP	save cursor position	
V-SGR	set latch graphics rendition	11

Video Processing Function Return Codes

When the video processing function returns 0, **JAM** will continue with normal processing. If it returns any other value, **JAM** will assume that the operation has been handled in the hook function. This allows the developer to implement only necessary operations.

Other Hook Functions

The Screen Manager provides an additional hook to handle block mode terminals. This function is best viewed as a driver. Block mode is described in Chapter 10.

2.3. CODING STRATEGY, RULES AND PITFALLS

2.3.1. Displaying Screens

There are a number of library functions provided for the display of screens as forms or windows. In general, the following rules and guidelines should be followed in choosing between them and deciding when they can be used:

- The display of screens as forms or windows from within screen functions at screen entry or screen exit is neither recommended nor supported.
- The routines `xsm_jform`, `xsm_jwindow`, and `xsm_jclose` are provided specifically for the display and destruction of screens in applications that use the **JAM** Executive. Applications not using the **JAM** Executive should not use these routines. They are recommended over

the other screen display routines in applications that do use the JAM Executive.

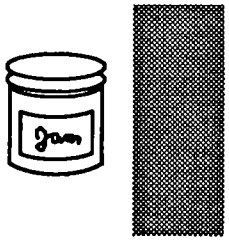
- The form display routine `xsm_jform` manipulates the form stack appropriately. The use of any other form display routines in applications that use the JAM Executive will exhibit unexpected behavior, as the form stack will not be synchronized with the application flow.

2.3.2.

Recursion

The developer should be careful, when using hook functions, to avoid the recursion that will come from nested hook function calls. Such recursion will not be easy to detect in the source code itself: some understanding of the product mechanism is required.

For example, care should be taken when writing record, playback, or key change functions that read from the keyboard, or status line functions that themselves cause the status line to be flushed. A default screen entry function that in and of itself opens new screens could be a problem.



Chapter 3.
Local Data Block

The Local Data Block, or LDB, is a region of memory for the storage of JAM field data that is generally shared between screens. It is discussed in the JAM Development Overview and in the Author's Guide.

3.1.
LDB CREATION

The LDB is created with the library routine call `xsm_ldb_init`. This routine searches for a data dictionary file created from the authoring tool with the Data Dictionary Editor. For more information about the data dictionary and the Data Dictionary Editor, see the Author's Guide.

If the data dictionary file is found, it is read and a single LDB entry is created in memory for every data dictionary entry that has a non–zero scope. Note that only the name of the LDB entry is placed in memory, storage for the field data that is stored with the entry is not allocated until the entry is used.

After it is created, the LDB is initialized from ASCII text files. These files, described in the Author's Guide, contain pairs of LDB names and values. The LDB entries named are filled with the values that follow them in the files.

3.2.
HOW JAM USES THE LDB

JAM uses the LDB for the storage and propagation of field data from screen to screen in the application. Every time a screen is opened, or exposed by the closing of a window that

covers it, every field on the screen named identically to an LDB entry is filled with the value of the LDB entry. This occurs after the screen entry function is called.

Correspondingly, every time a screen is closed, or hidden when a window pops up over it, every LDB entry that is named identically to a field on the screen is filled with the value of the screen field. This occurs before the screen exit function is called.

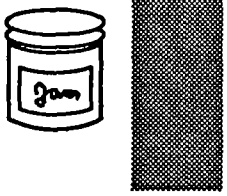
When a screen is populated from the LDB at screen entry time, there is a subtle difference between a new screen being opened and a screen being exposed when a covering window is closed. When a screen is newly opened, only empty fields with corresponding LDB entries will be populated from the LDB. When a screen is exposed, all fields that have corresponding LDB entries will be populated.

3.3. LDB ACCESS

Data in the LDB can be accessed with the library routines `xsm_n_getfield`, `xsm_n_putfield`, `xsm_i_getfield`, `xsm_i_putfield`, and related functions that access data by field name. These routines access the data on the current screen if the field that is named exists on the current screen. If the field does not exist on the current screen, these routines access the LDB.

During screen entry and exit processing only, the search order is reversed. During the screen entry and exit functions, these access routines first search the LDB and then search the screen. This is because the LDB is merged to the screen after the screen entry function, and the screen is stored to the LDB before the screen exit function. If the search order were not reversed the data accessed would be invalid[17].

17. This could, in a very small number of cases, introduce some incompatibilities with applications that were written with earlier releases of JAM. If such compatibility problems arise, use the library function `XSM_OPTION` setting the option `ENTEXT-OPTION` to `FORM-FIRST`.



Chapter 4.
Built–in Control Functions

This section describes control functions supplied with **JAM**. Note that the synopsis is for a **JAM** control string, not a programming language source statement. The return value of a control function can be used in a target list; see the Author's Guide for information on control strings and target lists.

You may use these functions in control strings and in JPL `call` statements.

jm_exit

end processing and leave the current screen

SYNOPSIS

```
^jm_exit
```

DESCRIPTION

Clears the current form or window and returns to the previous one. If the current form is the application's top–level form, **JAM** will prompt and exit to the operating system.

The effect is like the default action of the run–time system's EXIT key.

EXAMPLE

The following control string invokes a function named `process`. If it returns 0, another function is invoked to reinitialize the screen; but if it returns –1, the screen is exited. See `jm_gotop` for another example.

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

The example below shows how a form or a window can be replaced by another form or a window:

```
^(0=&w2)jm_exit
```

jm_gotop

return to application's top–level form

SYNOPSIS

```
^jm_gotop
```

DESCRIPTION

Returns to the application's top–level screen, ordinarily the first screen to appear when the application was run. All forms on the form stack and windows on the window stack are discarded.

The run–time system's SPF1 key performs the same action, unless you change it using SMINICTRL.

EXAMPLE

The following menu makes use of both `jm_exit` and `jm_gotop`.

```
+-------------------------------------------------+
:                                                 :
:  Query customer database__    custquery.jam__   :
:  Update customer database_    custupdate.jam_   :
:  Free-form query_________    !sql___________   :
:  Return to previous menu__    ^jm_exit_______   :
:  Return to main menu______    ^jm_gotop______   :
:                                                 :
+-------------------------------------------------+
```

jm_goform

prompt for and display an arbitrary form

SYNOPSIS

```
^jm_goform
```

DESCRIPTION

This function pops up a window in which you may enter the name of a form; it will then close all open windows and attempt to display the form, as if that form's name had appeared in a control string. It is useful for providing a shortcut around a menu system for experienced users.

The result is the same as the default action of the run–time system's SPF3 key.

EXAMPLE

The following line, if placed in your setup file, will make the PF10 key act like SPF3 normally does:

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

jm_keys
simulate keyboard input

SYNOPSIS

```
^jm_keys keyname-or-string (keyname-or-string ...)
```

DESCRIPTION

Queues characters and function keys that appear after the function name for input to the run–time system, using `xsm_ungetkey`. The run–time system then behaves as though you had typed the keys.

Function keys should be written using the logical key mnemonics listed in *smkeys*.incl.cobol . Data characters should be enclosed between apostrophes `' '`, back-quotes `` ` ` ``, or double quotes `" "`. This function passes its arguments to `xsm_ungetkey` in reverse order, so you supply them in the natural order.

`jm_keys` will process a maximum of 20 keys. This limit includes function keys plus characters contained in strings.

EXAMPLE

Enter the name of your favorite bar, followed by a tab and the name of its owner:

```
^jm_keys 'Steinway Brauhall' TAB "James O'Shaughnessy"
```

Return to the preceding menu and choose the second option:

```
^jm_keys EXIT HOME TAB XMIT
```

jm_mnutogl

switch between menu and data entry mode on a dual–purpose screen

SYNOPSIS

```
^jm_mnutogl {screen-mode}
```

DESCRIPTION

JAM supports the use of a single screen for both menu selection and data entry; one popular example is a data entry screen with a "menu bar". The screen must, however, be either one or the other at any given moment. This function switches the run–time system's treatment of the screen to the other mode. This function performs the same function as the MTGL logical key.

An optional argument may be specified which will force the screen into a particular mode, regardless of its current state. To specify menu mode, use the argument 'M' (or 'm'). To specify open–keyboard (data entry) mode, use the argument 'O' (or 'o').

jm_system

prompt for and execute an operating system command

SYNOPSIS

```
^jm_system
```

DESCRIPTION

This function pops up a small window, in which you may enter an operating system command. When you press TRANSMIT, it closes the window and executes the command. While the command is executing, your terminal is returned to the operating system's default I/O mode.

The run–time system's SPF2 key invokes this function by default.

EXAMPLE

The following line, when placed in your setup file, will cause the PF10 key to act as SPF2 normally does:

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

jm_winsize

allow end–user to interactively move and resize a window

SYNOPSIS

```
^jm_winsize
```

DESCRIPTION

Calling `jm_winsize` has the same effect as if the end–user had just hit the VWPT (viewport) logical key. The viewport status line appears and the user can move, resize and change the offset of the screen as well as move to any sibling windows. When the end–user hits XMIT (transmit) the previous status line is restored.

In order for the end–user to able to move from one window to another, the windows must be siblings. Windows may be specified as siblings by specifying `&&` in a JAM control string. See the sections on "Viewports and Positioning" and "Control Strings" in the Author's Guide for further information. This function parallels the library routine `xsm_winsize`.

jpl
invoke a JPL procedure

SYNOPSIS

```
^jpl procedure [ argument ... ]
```

DESCRIPTION

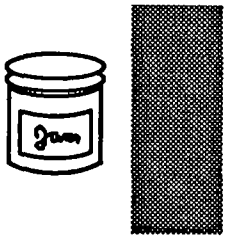
This function invokes a procedure written in the JYACC Procedural Language. procedure should be the name of a JPL procedure or module; anything following that will be passed to the procedure as arguments. See the JPL Programmer's Guide for the rules used by the JPL interpreter to determine which JPL procedure is executed. The value returned by your procedure will be returned by `jpl` for use in a target list.

This function is similar to the JPL `jpl` command. Colon expansion is done on the arguments.

EXAMPLE

The control string below invokes a JPL function to concatenate two strings and store the result in `target`.

```
^jpl concat target "king" "kong"
```



Chapter 5.
Keyboard Input

Keystrokes are processed in three steps. First, the sequence of characters generated by one key is identified. Next the sequence is translated to an internal value, or logical character. Finally, the internal value is either acted upon or returned to the application ("key routing"). All three steps are table–driven. Hooks are provided at several points for application processing; they are described in the chapter "Writing and Installing Hook Functions".

5.1.
LOGICAL KEYS

JAM processes characters internally as logical values, which frequently (but not always) correspond to the physical ASCII codes used by terminal keyboards and displays. Specific physical keys or sequences of physical keys are mapped to logical values by the key translation table, and logical characters are mapped to video output by the MODE and GRAPH commands in the video file. For most keys, such as the normal displayable characters, no explicit mapping is necessary. Certain ranges of logical characters are interpreted specially by JAM; they are

- `0x0100 to 0x01ff`: operations such as tab, scrolling, cursor motion
- `0x6101 to 0x7801`: function keys PF1 – PF24
- `0x4101 to 0x5801`: shifted function keys SPF1 – SPF24
- `0x6102 to 0x7802`: application keys APP1 – APP24

5.2.
KEY TRANSLATION

The first two steps together are controlled by the key translation table, which is loaded during initialization. The name of the table is found in the environment (see the configuration guide for details). The table itself is derived from an ASCII file which can be modified by any editor; a screen–oriented utility, `modkey`, is also supplied for creating and modifying key translation tables (see the Utilities Guide).

JAM assumes that the first character of any multi–character key sequence to be translated to a single logical key is a control character in the ASCII chart (0x00 to 0x1f, 0x7f, 0x80 to 0x9f, or 0xff). All characters not in this range are assumed to be displayable characters and are not translated.

Upon receipt of a control character, the keyboard input function `xsm_getkey` searches the translation table. If no match is found on the first character, the key is accepted without translation. If a full match is found on the first character, an exact match has been found, and `xsm_getkey` returns the value indicated in the table. The search continues through subsequent characters until either

1. an exact match on n characters is found and the n+1'th character in the table is zero, or n is 6. In this case the value in the table is returned.
2. an exact match is found on n–1 characters but not on n. In this case `xsm_getkey` attempts to flush the sequence of characters returned by the key.

This last step is of some importance: if the operator presses a function key that is not in the table, the Screen Manager must know "where the key ends". The algorithm used is as follows. The table 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 no entry matches by type at the n'th character, the shortest sequence that matches on n–1 characters is used.) This method allows `xsm_getkey` to distinguish, for example, between the sequences `ESC O x`, `ESC [ A`, and `ESC [ 1 0 ~`.

5.3.
KEY ROUTING

The main routine for keyboard processing is `xsm_input`. This routine calls `xsm_getkey` to obtain the translated value of the key. It then decides what to do based on the following rules.

If the value is greater than 0x1ff, `xsm_input` returns to the caller with this value as the return code.

If the value is between 0x01 and 0x1ff, the key is first translated via the key translation table. This table is changed with the library routine `xsm_keyoption`. Then processing is determined by a routing table. Use `xsm_keyoption` to get and set the routing information for a particular key. The routing value consists of two bits, examined independently, so four different actions are possible:

1. If neither bit is set, the key is ignored.
2. If the EXECUTE bit is set and the 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.
3. If the RETURN bit is set, `xsm_input` returns the logical value to the caller; otherwise, `xsm_getkey` is called for another value.
4. If both bits are set, the key is executed and then returned.

The default settings are *ignore* 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 exteneded ASCII characters are ignored. The application function keys (PF1–24, SPF1–24, APP1–24, 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.

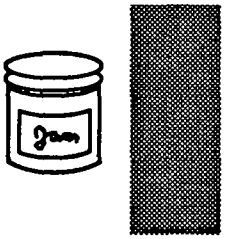
Applications can change key actions on the fly by using `xsm_keyoption`. For example, to disable the backtab key the application program would execute

```
call "xsm_keyoption" using BACK, KEY-ROUTING, KEY-IGNORE
```

To make the field erase key return to the application program, use

```
call "xsm_keyoption" using FERA, KEY-ROUTING, RETURN
```

Key values can be found in the file `smkeys.incl.cobol`.



Chapter 6.
Terminal Output Processing

JAM uses a sophisticated *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 being solicited or the library function `xsm_flush` has been called. Instead, the run–time system does screen updates in memory, and keeps track of the display positions thus "dirtied". Flushing begins when the keyboard is opened; but if you type a character while flushing is incomplete, the run–time system will process it before sending any more output to the display. This makes it possible to type ahead on slow lines. You may force the display to be updated by calling `xsm_flush`.

JAM takes pains to avoid code specific to particular displays or terminals. To achieve this it defines a set of logical screen operations (such as "position the cursor"), and stores the character sequences for performing these operations on each type of display in a file specific to the display. Logical display operations and the coding of sequences are detailed in the Video Manual; the following sections describe additional ways in which applications may use the information encoded in the video file.

6.1. GRAPHICS CHARACTERS AND ALTERNATE CHARACTER SETS

Many terminals support the display of graphics or special characters through alternate character sets. Control sequences switch the terminal among the various sets, and characters in the standard ASCII range are displayed differently in different sets. JAM supports alternate character sets via the `MODEx` and `GRAPH` commands 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 will first transmit the sequence that switches to `mode`, then transmit `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.

6.2. THE STATUS LINE

JAM reserves one line on the display for error and other status messages. Many terminals have a special status line (not addressable with normal cursor positioning); if such is not the case, JAM will use the bottom line of the display for messages. There are several sorts of messages that use the status line; they appear below in priority order.

1. Transient messages issued by `xsm_err_reset` or a related function
2. Ready/wait status
3. Messages installed with `xsm_d_msg_line` or `xsm_msg`
4. Field status text
5. Background status text

There are several routines that display a message on the status line, wait for acknowledgement from the operator, and then reset the status line to its previous state: `xsm_query_msg`, `xsm_err_reset`, `xsm_emsg`, `xsm_quiet_err`, and `xsm_qui_msg`. `xsm_query_msg` waits for a yes/no response, which it returns to the calling program; the others wait for you to acknowledge the message. These messages have highest precedence.

`xsm_setstatus` provides an alternating pair of background messages, which have next highest precedence. Whenever the keyboard is open for input the status line displays `Ready`; it displays `Wait` when your program is processing and the keyboard is not open. The strings may be altered by changing the `SM-READY` and `SM-WAIT` entries in the message file.

If you call `xsm_d_msg_line`, the display attribute and message text you pass remain on the status line until erased by another call or overridden by a message of higher precedence.

When the status line has no higher priority text, the Screen Manager checks the current field for text to be displayed on the status line. If the cursor is not in a field, or if it is in a field with no status text, JAM looks for background status text, the lowest priority. Background status text can be set by calling `xsm_setbkstat`, passing it the message text and display attribute.

In addition to messages, the rightmost part of the status line can display the cursor's current screen position, as, for example, `C 2,18`. This display is controlled by calls to `xsm_c_vis`.

During debugging, calls to `xsm_err_reset` or `xsm_quiet_err` can be used to provide status information to the programmer without disturbing the main screen display. Keep in mind that these calls will work properly only after screen handling has been initialized by a call to `xsm_initcrt`. `xsm_err_reset` and `xsm_quiet_err` can be called with a message text that is defined locally, as in:

```
move "Zip code invalid for this state" to message.
call "xsm_err_reset" using message.
```

However, the JAM library functions use a set of messages defined in an internal message table. This table is accessed by the function `xsm_msg_get`, using a set of defines in the header file `smerror.incl.cobol`. The return value from `xsm_msg_get` can be used as input for one of the status line functions.[18]

The message table is initialized from the message file identified by the environment variable SMMSGS. Application messages can also be placed in the message file. See the section on message files in the Configuration Guide.

18. See Appendix B. for specific message numbers and their meanings.

Chapter 7.
Writing International (8 bit) Applications

7.1. INTRODUCTION

This chapter describes how to use the 8 bit internationalization capabilities that have been incorporated into JAM Release 5.

From the point of view of someone who has used JAM without these features, a few differences will be apparent immediately. Other, more subtle, differences will emerge as the package is used in building language–independent applications. Finally, many of the changes were made so that the development utilities could be localized for use in other countries. These will largly go unnoticed by people using the package in English.

7.1.1. General Overview

The purpose of the 8 bit NLS is to allow the JAM product and applications created with with it to be "localized" for use in non–English–speaking countries. This means that the product can be made to look like it originated in 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. Notwithstanding this, many of the features that are only visible to programmers will continue to be in English since many programmers are used to working in English.

The capabilities described are limited to languages in which characters can be represented in 8 bits of information and those that use a left–to–right entry order. This eliminates the complexities associated with many far– and middle–eastern languages.

7.2. LOCALIZATION

JAM and JAM applications can be localized by taking the following steps:

- Use the Screen Editor to translate all screens in the application.
- Modify and recompile the message file.
- Translate the documentation.

7.2.1. Background

The JAM product was originally developed with some internationalization issues in mind. It has always used 8 bit character data, without appropriating a bit for internal use. So one of the major demands of the international market was already satisfied.

Date and time formats have always been completely specified by the screen creator. The wide variety of formats available in Release 4 could satisfy most requirements. In Release 5, additional capabilities were added to make it easier to convert screens from one language to another. Currency formats were the least international of the features in the Release 4 product. Release 5 makes these completely language independent.

Each of the sections below discusses some aspect of internationalization.

7.2.2. 8 Bit Character Data

As pointed out in the introduction, JAM supports 8 bit character data. Video files specific to the terminal can give special instructions, if necessary, as to 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 will not need to shift character sets.

The video file can also be used to translate between two different standards for international characters. Thus the screens could 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 character set (e.g. 0x01 – 0x1f, or 0x80 – 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. The cursor control keys (arrows, tab, etc.), function keys, and soft 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 will 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, `modkey` (or a text editor) would be used to map the byte sequences into a logical value, just as the video file would be used to map the logical value to the sequence required by the display terminal.

If you have questions about how to display non–English characters or to receive them from the keyboard, consult the chapters on keyboard and video processing.

7.2.3.
Date And Time Fields

Date and Time fields have been completely revamped in Release 5. They have been combined to enable one field to have both date and time information. This, and the fact that more flexibility was added to date and time formatting, required changes to the date and time mnemonics. For example, in Release 4, the mnemonic `mm` was used for a 2–digit month in Date fields as well as the specifier for minutes in Time fields. Clearly, this cannot serve both purposes when the fields are combined.

In Release 5, the mnemonics for specifying date and time formats are stored in the message file so they may be changed. In addition, they are stored in a "tokenized" form internally which provides two major benefits. First, the need to parse the formats at runtime is eliminated, thus speeding up processing and reducing memory requirements. Second, screen designers in different countries editing the same screen will all see date and time specifications in formats they are used to. For example, if an English screen designer created a date field with the format `mon/day/year`, it might show up on a French system as `mois/jour/annee`.

The problem of interchanging the month and day is dealt with later.

The table below shows the default message file entries for date and time mnemonics:

Msg # Mnemonic	*Date/Time Mnemonic*	*Tokenized Format*	*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	abbrev. month name
FM_MONL	MONL	%*m	long month name

Thus, a date field specified as mm/dd/yyyy in Release 4 would be MON2/DATE2/YR4 in Release 5. The f4to5 conversion program will convert the format to %m/%d/%4y internally so it will automatically show up correctly when the screen is edited. The mnemonics were chosen to correspond to ANSI standards. You can change them to suit your own needs by simply changing the message file and running msg2bin. To change the mnemonic for a 4 digit year from YR4 to YYYY, for example, change the message file line

```
FM_YR4 = YR4
```

to

```
FM_YR4 = YYYY
```

and run `msg2bin`.

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. What is important, however, is to be able to modify an application to operate in a different language. The goal is that only the text of the screens and the message file should need to be changed.

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

```
Mon Apr 4, 1989
```

whereas on a French system it would be

```
Lun Avr 4, 1989
```

This happens without changing the date format. All that has changed are the names and abbreviations of the months and days which are also stored in the message file so it is a simple matter to convert them.

Now consider a date field which in English should show up in `mm/dd/yyyy` form but should appear in French as `dd-mm-yyyy`. In this case, the date format itself would have to be modified. For this reason, 10 additional formats are supplied for the designer's use. For instance, in the message file the designer can specify a new date mnemonic called `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 screen, needs to be changed to convert the date field to French.

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.

The following table shows the default message file entries for these extra date mnemonics:

ATT–Msg Number Mnemonic	ATT–Date/Time Mnemonic	ATT–Tokenized Form	ATT–Corresponding Msgfile Entry	ATT–Default
ATT–FM_0MN_DEF_DT	ATT–DEFAULT	ATT–%0f	ATT–SM_0DEF_DTIME	ATT–%m/%d/%2y %h:%0M
ATT–FM_1MN_DEF_DT	ATT–DEFAULT DATE	ATT–%1f	ATT–SM_1DEF_DTIME	ATT–%m/%d/%2y
ATT–FM_2MN_DEF_DT	ATT–DEFAULT TIME	ATT–%2f	ATT–SM_2DEF_DTIME	ATT–%h:%0M
ATT–FM_3MN_DEF_DT	ATT–DEFAULT3	ATT–%3f	ATT–SM_3DEF_DTIME	ATT–%m/%d/%2y %h:%0M
ATT–FM_4MN_DEF_DT	ATT–DEFAULT4	ATT–%4f	ATT–SM_4DEF_DTIME	ATT–%m/%d/%2y %h:%0M
ATT–FM_5MN_DEF_DT	ATT–DEFAULT5	ATT–%5f	ATT–SM_5DEF_DTIME	ATT–%m/%d/%2y %h:%0M
ATT–FM_6MN_DEF_DT	ATT–DEFAULT6	ATT–%6f	ATT–SM_6DEF_DTIME	ATT–%m/%d/%2y %h:%0M
ATT–FM_7MN_DEF_DT	ATT–DEFAULT7	ATT–%7f	ATT–SM_7DEF_DTIME	ATT–%m/%d/%2y %h:%0M
ATT–FM_8MN_DEF_DT	ATT–DEFAULT8	ATT–%8f	ATT–SM_8DEF_DTIME	ATT–%m/%d/%2y %h:%0M
ATT–FM_9MN_DEF_DT	ATT–DEFAULT9	ATT–%9f	ATT–SM_9DEF_DTIME	ATT–%m/%d/%2y %h:%0M

Thus, if the screen designer specifies a date field with the format DEFAULT DATE, it would show up in mm/dd/yy form. If the line

```
SM_1DEF_DTIME = %m/%d/%2y
```

in the message file were changed to

```
SM_1DEF_DTIME = %d-%m-%2y
```

the date would show up in dd-mm-yy form. To change the mnemonic for this date format to REGULAR DATE, the message FM_1MN_DEF_DT should be modified.

7.2.4.

Currency Fields

Like Date and Time fields, Currency fields have been modified in Release 5. Since it is not uncommon in Europe to be dealing with several currencies simultaneously, release 5 does not force any one system on the screen creator. Thus, the formatting capabilities were enhanced to support any convention the screen creator might desire. As with date and time formats, a "default" format is supplied that causes the actual format to be taken from the message file. For Currency fields however, this option is supplied only for the parts of the format that may vary from one currency to another.

The new release allows the following items to be specified for Currency fields:

- the decimal symbol (usually dot or comma)
- minimum number of decimal places
- maximum number of decimal places
- thousands separator (usually dot or comma; b = blank)
- the currency symbol to be used (up to 5 characters)
- the placement of that symbol (left, right or at decimal pt)
- default currency from the message file (to replace the above entries)
- rounding (round–up, round–down, round–adjust)
- fill character
- justification
- clear if zero
- apply if empty

There is a slight problem in specifying currency symbols when using the Screen Editor. Since the currency symbol is entered into a regular field, it is not possible to enter trailing

spaces (they are always stripped off). Thus, to specify a leading currency symbol separated from the data by a space (`FF 123.456,78`) you must use the message file. For this reason, the dot (`.`) may be used to signify a space when entered into the currency field. A dot in the message file for this purpose will appear as a dot.

The default currency formats are strings of the form ***rmxtpccccc*** where:

- ***r*** = decimal symbol (usually comma or dot)
- ***m*** = minimum number of decimal places
- ***x*** = maximum number of decimal places
- ***t*** = thousands separator (usually comma or dot; b = blank)
- ***p*** = placement of currency symbol (l, r or m)
- ***ccccc*** = up to 5 characters for the currency symbol

Thus, if the screen designer specifies a currency field with the format `CURRENCY`, it would show up in `$999,999.99` form. If the line

```
SM_0DEF_CURR = ".22,1$"
```

in the message file were changed to

```
SM_0DEF_CURR = ",22.1FF"
```

the field would show up as `FF 999.99,99`. To change the mnemonic for this currency field, the message `FM_0MN_CURRDEF` should be modified. The following table shows the default message file entries for the currency mnemonics:

Msg Number Mnemonic	*Currency Mnemonic*	*Corresponding Msgfile Entry*	*Default*
FM_0MN_CURRDEF	CURRENCY	SM_0DEF_CURR	.22,1$
FM_1MN_CURRDEF	NUMERIC	SM_1DEF_CURR	.09,
FM_2MN_CURRDEF	PLAIN	SM_2DEF_CURR	.09
FM_3MN_CURRDEF	DEFAULT3	SM_3DEF_CURR	.09
FM_4MN_CURRDEF	DEFAULT4	SM_4DEF_CURR	.09
FM_5MN_CURRDEF	DEFAULT5	SM_5DEF_CURR	.09
FM_6MN_CURRDEF	DEFAULT6	SM_6DEF_CURR	.09
FM_7MN_CURRDEF	DEFAULT7	SM_7DEF_CURR	.09

Msg Number Mnemonic	*Currency Mnemonic*	*Corresponding Msgfile Entry*	*Default*
FM_8MN_CURRDEF	DEFAULT8	SM_8DEF_CURR	.09
FM_9MN_CURRDEF	DEFAULT9	SM_9DEF_CURR	.09

7.2.5.

Decimal Symbols

JAM 5 will accomodate 3 decimal symbols which are used in different circumstances:

- System Decimal Symbol
- Local Decimal Symbol
- Field Decimal Symbol

The System Decimal Symbol is the one that library routines like `atof` and `sprintf` use. The Local Decimal Symbol is the one that is used when local customs are followed (dot in English; comma in French). The Field Decimal Symbol is the one specified for a given field if that field is not observing local conventions.

The System and Local Decimal Symbols are obtained from the operating system if the operating system supports such things (see the installation notes for JAM for your operating system). The Local Decimal Symbol may be specified in the message file (message `SM_DECIMAL`), in which case it overrides the operating system decimal symbol. Dot is the system decimal if no symbol is specified in the message file and if the operating system does not supply one.

The sections below describe the circumstances under which each of the different symbols is used.

7.2.6.

Character Filters

The one time that JAM requires some knowledge of the meaning of the data is while enforcing the character filters on a field. The filters currently supported are digits only, numeric, alphabetic, alphanumeric, and yes/no and regular expression.

To validate the data JAM uses the standard C macros: `isdigit`, `isalpha`, etc. JAM 5 assumes that the operating system supplies these macros in a form suitable for interna-

tional use. In 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 will supply information about these characters, the proposed ANSI standard does not address the issue. Therefore, for reasons of portability, JAM will continue 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 5 uses the ASCII collating sequence for ranges of characters. Therefore, the expression

```
[a-z]*
```

will match only the English lower case letters. The European character ä, for example, would not be matched by this expression.

7.2.7.
Status And Error Messages

All messages produced by JAM 5 are stored in the message file so they may be easily localized. Each message is a complete phrase or sentence. Message components are never pieced together because doing so would make it difficult to translate to a language that has a sentence structure different from English.

7.2.8.
Screens In The Utilities

These screens were memory resident in Release 4. For international customers they must be modifiable.

A linkable `jxform` is be provided, and the library containing the source for the screens is made available. A developer may translate the screens and relink the utilities. Similari-

ly `modkey` is developer–linkable, and the source for its screens is provided. In this way the screens remain memory resident and no compromise of speed need be made.

Unfortunately this solution is not ideal if several users on the same machine wish to use different languages. To support this, the screens may be kept on disk. The current mechanism of `SMPATH` allows run–time selection of the set of screens to be used.

7.2.9.
Screens In Application Programs

The same approach as discussed in the above section can be used for screens in application programs. Thus different language screens can be kept in separate directories and the user can specify which is to be used at run–time.

7.2.10.
Menu Processing

`xsm_input` returns the first character of the selected entry. This, of course, is not language independent. JAM utilities have been modified to use the current field number rather than the return value. Because it cannot be assumed that all entries will have unique first letters, the `string` option is specified.

Application programs intended for an international market should not rely on the initial character of the menu selection. The field number containing the cursor is a better way of determining which selection the operator has made. However the field numbers may change if the screen is redesigned. Note that this is not a problem when the JAM Executive is used, since the JAM Executive uses relative field numbers to determine the control string to execute when a menu field is selected.

A new additional edit was instituted in JAM 4 that specifies the return code from a `return entry` (or `menu`) field. The screen creator specifies the return code (an integer) when designing the screen. If this edit exists, `xsm_input` uses that value as the return code to the calling program. If this edit does not exist, the usual return code is used.

7.2.11.
`lstform`, `lstdd`, and `jammap`

These utilities list data about the screen in English. Since they are often used for documentation it is important that the text be translatable to other languages. Thus the textual material, headings, etc., have been moved to the message file.

7.2.12.

Range Checks

Range checks for numeric data are presently correctly handled since they use `atof` (assuming that the "strip" routine works properly).

Alphabet data presents special problems. 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 compare equal. Also, in a telephone book, `St.` and `Saint` compare equal, 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 given 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.

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.

For these reasons no changes have been made to the Release 4 method of range checking. `strcmp` and `memcmp` continue to be used. These compare the internal values of the characters, without regard to their meanings in the local language.

7.2.13.

Calculations Using `@SUM` and `@DATE`

These keywords have been retained even though they are language specific. Computations with dates assume the Gregorian calendar. No provison is made for other calendars.

7.2.14.

`xsm_dblval` and `xsm_dtofield`

These routines use `atof` and `sprintf` therefore correctly interpret the System Decimal Symbol (radix character).

7.2.15.

`xsm_is_yes` and `xsm_query_msg`

These routines use the characters in the message file for `y` and `n` and thus are already internationalized. They use `toupper` to recognize the upper case variations.

7.2.16.

Batch Utilities

All the utilities messages, including usage messages have been moved to the message file.

The mnemonics for logical keys (XMIT, EXIT, etc.) are not translated to other languages, nor the mnemonics used in the video file, so the internal processing of the utilities need not be modified.

Chapter 8.
Writing Portable Applications

The following section describes features of hardware and operating system software that can cause JAM to behave in a non–uniform fashion. An application designer wishing to create programs that run across a variety of systems will need to be aware of these factors.

8.1.
TERMINAL DEPENDENCIES

JAM can run on display terminals of any size. On terminals without a separately addressable status line, JAM will steal the bottom line of the display (often the 24th) for a status line, and status messages will overlay whatever is on that line. A good lowest common denominator for screen sizes is 23 lines by 80 columns, including the border (21 if two–line soft key labels will be used).

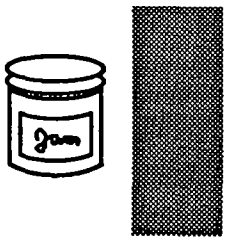
Different terminals support different sets of attributes. JAM makes sensible compromises based on the attributes available; but programs that rely extensively on attribute manipulation to highlight data may be confusing to users of terminals with an insufficient number of attributes. Colors will not show up on monochrome terminals, e.g. Use of graphics character sets is particularly terminal dependent.

Attribute handling can also affect the spacing of fields and text. In particular, anyone designing screens to run on terminals with onscreen attributes must remember to 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).

The key translation table mechanism supports the assignment of any key or key sequence to a particular logical character. However, the number and labelling of function keys on particular keyboards can constrain the application designer who makes heavy use of func-

tion keys for program control. The standard VT100, for instance, has only four function keys. For simple choices among alternatives, menus are probably better than switching on function keys.

Using function key labels, or keytops, instead of hard–coded key names is also important to making an application run smoothly on a variety of terminals. Field status text and other status line messages can have keytops inserted automatically, using the %K escape. No such translation is done for strings written to fields; in such cases, you may want to place the strings in a message file, since the setup file can specify terminal–dependent message files.



Chapter 9.
Writing Efficient Applications

9.1.
MEMORY–RESIDENT SCREENS

Memory–resident screens are much quicker to display than disk–resident screens, since no disk access is necessary to obtain the screen data. However, the screens must first be converted to source language modules with `bin2cob` or a related utility (see the Utilities Guide), then compiled and linked with the application program.

`xsm_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 will generally have the same name as the file the original screen was originally stored in.

Using memory–resident screens (and configuration files, see the next section) is, of course, a space–time tradeoff: increased memory usage for better speed.

9.2.
MEMORY–RESIDENT CONFIGURATION FILES

Any or all of the three configuration files required by **JAM** can be made memory resident. First a COBOL source file must be created from the binary version of the file, using the `bin2cob` utility; see the Utilities Guide. The source files created are not readily decipherable. Then the COBOL source is included in a cobol program via the `COPY` state-

ment. A call is then made to either `xsm_msgread`, `xsm_vinit`, or `xsm_ keyinit`, depending on the type of configuration file being installed.

If a file is made memory–resident, the corresponding environment variable or `SMVARS` entry can be dispensed with.

9.3. MESSAGE FILE OPTIONS

If you need to conserve memory and have a large number of messages in message files, you can make use of the MSG_DSK option to `xsm_msgread`. This option avoids loading the message files into memory; instead, they are left open, and the messages are fetched from disk when needed. Bear in mind that this uses up additional file descriptors, and that buffering the open file consumes a certain amount of system memory; you will gain little unless your message files are quite large.

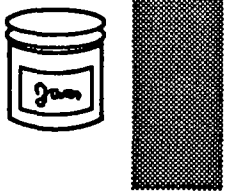
9.4. AVOIDING UNNECESSARY SCREEN OUTPUT

Several of the 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. To take an example: JAM can do all its cursor positioning using the CUP (absolute cursor position) command. However, it will use the 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 is capable of saving and restoring the cursor position itself (SCP, RCP), JAM will use those sequences instead of the more verbose CUP.

The global variable `I_NODISP` may also be used to decrease screen output. While this variable is set to 0 (via `xsm_iset`), calls into the JAM library will cause the internal screen image to be updated, but nothing will be written to the actual display; the display can be brought up to date by resetting `I_NODISP` to 1 and calling `xsm_rescreen`. With the implementation of delayed write this sort of trick is rarely necessary.

9.5. JPL VS. COMPILED LANGUAGES

JPL code execution goes through an extra layer of intrepretation that compiled code, such as COBOL, does not. In most cases, the total run time is too small to matter, but if a JPL function is long or loops many times and a delay is noted, it may pay to rewrite it in COBOL.



Chapter 10.
Block Mode

The purpose of this document is to describe the block mode capabilities of JAM from the perspective of someone using the system and from the perspective of a developer that needs to write a block mode driver.

10.1.
USING BLOCK MODE

10.1.1.
General Overview

The purpose of the block mode interface is to allow JAM to be used with terminals, like the HP2392A and IBM 3270's, that operate in block mode. Such terminals, which are hereinafter referred to as block mode terminals, operate differently than their interactive or character mode counterparts in that they do not interact with the computer on every keystroke. Instead, a formatted screen is sent to the terminal and processed by the terminal locally. When a function key is pressed, data are transmitted to the computer and are available to the program which sent the formatted screen.

Block mode terminals typically have capabilities for defining protected and unprotected fields and sometimes allow a minimal set of character validations such as restricting a field to only allow digits. They do not provide JAM–like capabilities such as shifting, scrolling and provisions for post–field validation. It should therefore seem obvious that an application will behave slightly differently on a block mode terminal than on an interactive one. The goal of the block mode interface, however, is to minimize these differences and, to the greatest extent possible, allow applications to be created that can operate

in either mode without the need for the programmer to consider the differences. This is in keeping with the JAM philosophy of creating terminal–independent applications.

10.1.2.
Authoring

Certain JAM utilities, like `modkey`, the Screen Editor, and the Data Dictionary Editor only work in interactive mode. Thus, they can only be used with interactive terminals or those that can be switched programmatically between block and interactive mode.

`jxform` is the JAM authoring utility. It allows the user to navigate through the screens in an application and to invoke the Screen and Data Dictionary Editors when appropriate. When used with block mode–only terminals, `jxform` does not permit entry into the aforementioned utilities. When used with hybrid terminals (i.e. those that can switch between block and interactive mode programmatically), `jxform` forces interactive mode before entering the utilities.

10.1.3.
Selecting Block Mode

JAM operates with three types of terminals: interactive–only, block mode–only, and hybrid. Block mode can be used with either of the latter two.

By default, JAM operates in interactive mode regardless of the terminal type. To operate in block mode requires a block terminal driver to be linked with the system. (Block terminal drivers are described in detail later.) This alone, however, will not initiate block mode; two additional things must be done.

First there must be a call to `xsm_blkinit`. This is generally done in the "main" routine of the application, `jmain.cobol`. If this call is absent, the application will be run in interactive mode. Also the additional code to support block mode will not be linked with the program. Thus programs not desiring block mode support are not penalized.

Second the correct block mode driver must be selected. This can be done in one of two ways.

If the application program author knows the correct driver he/she can install it by calling `xsm_uinstall`. This should be done before calling `xsm_blkinit`. Typically the program will install a "hard–coded" driver, but it could instead key off of SMTERM, or some other environment variable, to find the correct one. In this case the application will run in block mode, independent of the end user's preference.

The second method for selecting the driver leaves the job to the end user. If `xsm_blkinit` is called without previously installing a driver, the entry BLKDRIVER in the video

file is examined. If it is absent, `xsm_blkinit` fails and the application remains in interactive mode. If it is present the name given there is used to find the correct driver. This is done by a table lookup in a source routine (`blkdrvr.c`) that must be linked with the application. Naturally all possible choices of the driver must also be linked with the program. In this case the end user can override the application programers desire to use block mode.

The design allows for three scenarios: the programmer can prohibit block mode (no call to `xsm_blkinit`), the programmer can force block mode (`xsm_install` followed by `xsm_blkinit`), or the programmer can permit block mode but allow the end user final say (`xsm_blkinit` only).

Note that the application never calls `sm_blkdrvr`. The source code to that routine is given to customers to enable them to extend the capabilities of the second method.

10.1.4.

Differences Between Block Mode And Interactive Mode

Although every attempt has been made to preserve the look and feel of applications operating in block mode, the following differences between block mode and interactive mode should be noted.

Windows

Windows work much as they do in interactive mode. The only noticable difference is that the cursor is not be restricted to the active window as this is not possible in block mode. In keeping with the concepts of interactive mode, however, only the fields on the active window are unprotected.

Menus

In interactive mode, menus utilize a "bounce bar" to track the cursor. The bounce bar moves when cursor–positioning keys are pressed and when ascii data are typed. Since block mode terminals do not return these keys, another approach must be taken. We supply two options:

In option 1, menu fields in block mode are unprotected, making it easy for an operator to tab to them. To make a selection, the operator positions to the appropriate field and presses XMIT. Thus, selection is similar to interactive mode except there is no bounce bar and there is no provision for selecting by typing the first N characters of the menu choice.

If the operator inadvertently types over a menu field there are no adverse consequences as JAM will "remember" the contents and restore it at an appropriate time.

This approach works well since the same screens can be used for block and interactive mode operation. However, for those who do not wish to allow the operator to type over menu choice fields, option 2 may be chosen. With option 2, JAM creates an unprotected field to the left of each menu choice so the menu fields themselves can remain protected. The operator can tab to these new fields to make a selection, or type the first character of a menu field and press XMIT. The new fields to the left of the menu choices are created as long as there is room on the screen even if it means they would be placed in a border or a separate window. If there is no room on the screen because the menu field starts in position 1 or 2, the system reverts to option 1.

The above works well for traditional menus, but two–level (pull– down) menus pose a different problem in that the ONLY way to move horizontally in interactive mode is via the arrows (since TAB moves between the entries of the sub–menu). Thus, in block mode the following happens. When a pull–down menu is active, JAM unprotects all main menu fields except the one with which the pull–down is associated. Thus, the operator can either make a selection from the pull–down or tab to another main menu choice and press XMIT causing its sub–menu to be activated.

The two options for processing menus described above work equally well for pull–down menus.

Character Validation

The block mode interface takes advantage of the terminal's capabilities for character validation. However, for situations in which the specified validations go beyond what the terminal can handle, JAM will validate the character data during Screen Validation. The result will be something like this:

The operator enters alphabetic data in a digits–only field. When the XMIT key is pressed, all fields are validated in the normal fashion, left–to–right, top–to–bottom. Thus, the cursor will be positioned to the errant field and a message displayed.

Since programs do not rely on data being correct unless and until Screen Validation completes without error, this should pose no problem. The only consideration is that invalid character data can get into the screen buffer and LDB if the operator enters incorrect characters and then presses something like EXIT (this cannot happen in interactive mode because the invalid characters would not be allowed in the first place.

The only reason for mentioning this has to do with how punctuation characters in digits–only fields are handled. Let's say that a digits–only field got filled with slash ("/") characters and this, in turn, got transferred to the screen buffer and hence to the LDB. On a subsequent attempt to enter data into the field, an attempt to merge the slashes with the

entered data would be made. But since the field has ALL slash characters, there would be no room for the digits.

Thus, to eliminate the possibility of "punctuation character creep", when reading data from a digits–only field, JAM first strips out all punctuation characters from the field and then merges in the punctuation characters from the screen buffer.

Field Validation

Clearly, fields are not validated when TAB and RETURN are pressed as in interactive mode. Thus, like character validations, field validations will be deferred until Screen Validation. This should not be a problem since, even in interactive mode, the operator can usually bypass field validation by using the arrow keys to move from field to field. Therefore, programs should not rely on the data until Screen Validation passes without error in either mode.

One type of field validation is worth noting. Consider a field with an attached function which does a database lookup and displays information in another field. In interactive mode, this would usually be executed when the field is completed, so the user would see the result. Since this is not really a validation, deferring it until Screen Validation would not help because the data would never be seen by the operator. Therefore, if this type of feature is contemplated in a block mode environment, the database lookup should be attached to a function key rather than as an attached function.

Screen Validation

Screen validation works the same in interactive and block mode. The cursor will be positioned to the first field in error and a message will be displayed to the operator.

Right Justified Fields

Unless the block mode terminal supports this feature directly, the cursor will always be positioned to the left side of right justified fields when the cursor enters them.

Field Entry Function, Automatic Help, Status Text, etc.

These are disabled in block mode since JAM does not know when fields are entered.

Currency Fields

Currency edits are usually applied to fields as they are exited. In block mode, since this is not possible, currency formatting is done during screen validation. Care should be taken

with right justified currency formats since subsequent entry may be difficult for the reasons cited above in the section on right justified fields.

Shifting Fields

Normally fields shift when the left or right arrows are pressed with the cursor at the start or end of a shifting field or, in the case of unprotected fields, when the operator types off the edge of the field. Since arrows and data entry keys are not returned in block mode, this is not possible. To utilize shifting fields in block mode, use the logical keys: Shift Left and Shift Right. These shift the field by the shifting increment and work equally well in block and interactive mode.

An alternative is to use the Zoom feature if all shifting fields are limited to the width of the screen.

Scrolling Fields

This is similar to the situation with shifting fields. In block mode, one can define function keys as PAGE UP and PAGE DOWN, or use the Zoom feature.

Messages

Error messages are normally acknowledged by pressing the space bar, although the specific key used can vary depending on the setting of error message options. Also, options govern whether the key should be used as the next keystroke or discarded after the message is acknowledged. In block mode, ANY key that gets transmitted from the terminal will suffice to acknowledge messages, regardless of what key is defined for that purpose. Using or discarding the acknowledgement key apply equally to block mode and interactive mode.

With query messages, JAM normally expects a Y or N response. In block mode, JAM will create a field on the status line into which the Y or N response can be entered. This entry must be followed by the XMIT key for it to be accepted. On terminals that have a separate stauts line it is not possible to create such a field. In these cases, XMIT will be treated as a positive response; EXIT will be treated as a negative response.

Insert Mode

Insert mode will operate in whatever way the block mode terminal supports. However, since JAM never knows if insert mode is set or not in block mode, it will, for terminals in which this is a problem, reset insert mode before transmitting data to the terminal. This is so the new data will not be INSERTED into the terminal buffer, causing all other data to move around.

Non–Display Fields

If the block mode terminal supports this feature, it will be used.

System Calls

These operate as in interactive mode. However, before passing control to the OS, JAM sets the terminal to the mode (block or interactive) expected by the OS, and resets it upon return from the system call. The JAM routines `xsm_leave` and `xsm_return` do the same.

Zoom

With the exception of the limitations expressed in the sections on shifting and scrolling, Zoom works as in interactive mode.

Help and Item Selection

With the exception of the limitations expressed in the sections on shifting, scrolling, field entry and menu processing, these functions work as in interactive mode.

Groups

Radio buttons and check lists behave similar to menus as described above.

10.2.

WRITING A BLOCK MODE DRIVER

10.2.1.

Installation

There are two parts to the installation process. These were discussed in greater detail above.

First a block terminal driver must be installed. This driver performs the low level communication between JAM and the terminal. The COBOL interface does not currently support writing your own block mode drivers.

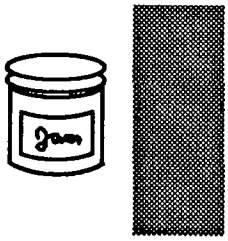
Next the application program must initiate block mode by making the appropriate subroutine call. The application program can also switch to interactive mode by means of a call. The assumption is that the default is interactive mode, thus a call to set block mode is needed even if that is the normal mode of the operating system. The application program can also set some operating parameters by means of a subroutine call.

10.2.2.

Application Program Support

JAM programs assume that the terminal is in interactive mode. Explicit calls are needed to switch from interactive to block and vice versa. To turn on block mode, the program should call `xsm_blkinit`. To turn off block mode (and turn on interactive mode) the program calls `xsm_blkreset`. The Screen Editor The key mapping utility (`modkey`) also requires interactive mode. The authoring utility (`jxform`) can be made to work in block mode, switching to interactive mode when the Screen Editor is invoked. This can be done by inserting the appropriate calls in `jxmain.cobol` (provided) and relinking `jxform`.

The routine `xsm_option` can be used to set some user–preference items.



Chapter 11.
Library Function Overview

In this chapter, we summarize the JAM library functions and list them in categories. All JAM library function names begin with the prefix `xsm_`. However, in the Function Reference Chapter and in this chapter, the functions are listed without prefix for clarity.

In addition to stripping off the prefix in the listings that follow, groups of closely related variant functions are listed under a single root name. The functions `xsm_r_form`, `xsm_d_form`, and `xsm_l_form`, for example, are all grouped under the heading `form`. In a few cases, functions may be listed under a name that is not a portion of the the function name but is suggestive of the utility of the function. For example, the function `xsm_r_at_cur`, which displays a window at the cursor position, is listed under the root name `window`, along with `xsm_r_window` (which displays a window at a fixed location) and a number of other window display routines. The calling syntax of each function is found in the SYNOPSIS section of the function listing in the Function Reference Chapter.

Most JAM library routines fall into one of the following categories:

- Initialization/Reset
- Screen and Viewport Control
- Keyboard and Display I/O
- Field/Array Data Access
- Field/Array Characteristic Access
- Group Access
- Local Data Block Access
- Cursor Control
- Message Display

- Scrolling and Shifting
- Mass Storage and Retrieval
- Validation
- Global Data and Changing JAM's Behavior
- Soft Keys and Keysets
- JAM Executive Control
- Block Mode Control
- Miscellaneous

The following sections summarize the functions that fall into these categories. Some listings are found in more than one category.

11.1. INITIALIZATION/RESET

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

`cancel`	reset the display and exit
`dicname`	set data dictionary name
`ininames`	record names of initial data files for local data block
`initcrt`	initialize the display and JAM data structures
`keyinit`	initialize key translation table
`ldb_init`	initialize (or reinitialize) the local data block
`leave`	prepare to leave a JAM application temporarily
`msgread`	read message file into memory
`resetcrt`	reset the terminal to operating system default state
`return`	prepare for return to JAM application
`vinit`	initialize video translation tables

11.2. SCREEN AND VIEWPORT CONTROL

The following routines are used to control viewports, the display of screens, and the form and window stacks.

close_window	close current window
form	display a screen as a form
hlp_by_name	display help window
issv	determine if a screen in the saved list
jclose	close current window or form under JAM Executive control
jform	display a screen as a form under JAM control
jwindow	display a window at a given position under JAM control
mwindow	display a status message in a window
shrink_to_fit	remove trailing empty array elements and shrink screen
sibling	define the current window as being or not being a sibling window
submenu_close	close the current submenu
svscreen	register a list of screens on the save list
unsvscreen	remove screens from the save list
viewport	modify viewport size and offset
wcount	obtain number of currently open windows
wdeselect	restore the formerly active window
window	display a window at a given position
winsize	allow end–user to interactively move and resize a window
wselect	activate a window

11.3.
DISPLAY TERMINAL I/O

The following routines provide the interface to JAM terminal I/O.

bel	beep!
bkrect	set background color of rectangle
do_region	rewrite part or all of a screen line
flush	flush delayed writes to the display
getkey	get logical value of the key hit
input	open the keyboard for data entry and menu selection
keyfilter	control keystroke record/playback filtering
keyhit	test whether a key has been typed ahead

`keylabel`	get the printable name of a logical key
`keyoption`	set cursor control key options
`m_flush`	flush the message line
`rescreen`	refresh the data displayed on the screen
`resize`	dynamically change the size of the display
`ungetkey`	push back a translated key on the input

11.4.
FIELD/ARRAY DATA ACCESS

The following routines access the data in fields and arrays. Most routines in this section have a number of variants that perform the same task but reference the field to be accessed differently. In these cases, the calling syntax of the *major* variant is listed under the SYNOPSIS section of the listing in the Function Reference Chapter. All other variants are listed under the VARIANTS section.

Most field access routines have five variants, although some have fewer. The five possible variants are shown in the table below:

Variants of Functions That Access Fields

Prefix	*Example*	*Description*
`xsm_`	`call "xsm_intval" using fieldnum.`	Access a field via field number.
`xsm_n_`	`call "xsm_n_intval" using fieldname.`	Access a field (or an entire array) via field name. Access the LDB if there is no field on the screen.
`xsm_i_`	`call "xsm_i_intval" using fieldname, occurrence.`	Access an occurrence via field name and occurrence number. Access the LDB if there is no field on the screen.
`xsm_o_`	`call "xsm_o_intval" using fieldnum, occurrence.`	Access an occurrence via field number and occurrence number.
`xsm_e_`	`call "xsm_e_intval" using fieldname, element.`	Access an element via field name and element number.

`amt_format`	write data to a field, applying currency editing
`calc`	execute a math edit style expression
`cl_unprot`	clear all unprotected fields
`clear_array`	clear all data in an array
`dblval`	get the value of a field as a real number
`dlength`	get the length of a field's contents
`doccur`	delete occurrences
`dtofield`	write a real number to a field
`fptr`	get the content of a field
`getfield`	copy the contents of a field
`gwrap`	get the contents of a wordwrap array
`intval`	get the integer value of a field
`ioccur`	insert blank occurrences into an array
`is_no`	test field for no
`is_yes`	test field for yes
`itofield`	write an integer value to a field
`lngval`	get the long integer value of a field
`ltofield`	place a long integer in a field
`null`	test if field is null
`putfield`	put a string into a field
`pwrap`	put text to a wordwrap field
`strip_amt_ptr`	strip amount editing characters from a string

11.5. FIELD/ARRAY ATTRIBUTE ACCESS

The following routines access information about fields and arrays. Like the routines in the previous section on field and array data access, each of these routines generally have five distinct variants. See the discussion in the introduction to the previous section for more information on variants of JAM library functions that access fields.

`base_fldno`	get the field number of the first element of an array
`bitop`	manipulate validation and data editing bits
`chg_attr`	change the display attribute of a field

`cl_all_mdts`	clear all MDT bits
`dlength`	get the length of a field's contents
`edit_ptr`	get special edit string
`finquire`	obtain information about a field
`fldno`	get the field number of an array element or occurrence
`ftog`	convert field references to group references
`ftype`	get the data type and precision of a field
`gtof`	convert a group name and index into a field number and occurrence
`length`	get the maximum length of a field
`max_occur`	get the maximum number of occurrences
`name`	obtain field name given field number
`num_occurs`	find the highest numbered occurrence containing data
`protect`	protect an array
`sc_max`	alter the maximum number of items allowed in a scrollable array
`size_of_array`	get the number of elements
`tst_all_mdts`	find first modified occurrence in the screen

11.6. GROUP ACCESS

The following routines access groups, that is, radio buttons and check lists. Groups are made up of fields that have attributes and data in them, but groups in and of themselves are implemented as phantom fields which take up no screen real estate. The value of a group indicates the set of selected consituent fields, although it is not recommended that that value ever be accessed or modified directly with any of the field access routines discussed in the preceding sections.

The routines that follow are those that are recommended for accessing groups:

`deselect`	deselect a checklist occurrence
`ftog`	convert field references to group references
`gp_inquire`	obtain information about a group
`gtof`	convert a group name and index into a field number and occurrence

`cl_all_mdts`	clear all MDT bits
`dlength`	get the length of a field's contents
`edit_ptr`	get special edit string
`finquire`	obtain information about a field
`fldno`	get the field number of an array element or occurrence
`ftog`	convert field references to group references
`ftype`	get the data type and precision of a field
`gtof`	convert a group name and index into a field number and occurrence
`length`	get the maximum length of a field
`max_occur`	get the maximum number of occurrences
`name`	obtain field name given field number
`num_occurs`	find the highest numbered occurrence containing data
`protect`	protect an array
`sc_max`	alter the maximum number of items allowed in a scrollable array
`size_of_array`	get the number of elements
`tst_all_mdts`	find first modified occurrence in the screen

11.6. GROUP ACCESS

The following routines access groups, that is, radio buttons and check lists. Groups are made up of fields that have attributes and data in them, but groups in and of themselves are implemented as phantom fields which take up no screen real estate. The value of a group indicates the set of selected consituent fields, although it is not recommended that that value ever be accessed or modified directly with any of the field access routines discussed in the preceding sections.

The routines that follow are those that are recommended for accessing groups:

`deselect`	deselect a checklist occurrence
`ftog`	convert field references to group references
`gp_inquire`	obtain information about a group
`gtof`	convert a group name and index into a field number and occurrence

`isselected`	determine whether a radio button or checklist occurrence has been selected
`select`	select a checklist or radio button occurrence

11.7. LOCAL DATA BLOCK ACCESS

The following routines access the Local Data Block, or LDB. Note that any of the field data access routines that reference fields by name or name and occurrence number (eg `xsm_n` and `xsm_i_` variants) will access the LDB if the named field does not exist on the active screen.

`allget`	load screen from the LDB
`dicname`	set data dictionary name
`dd_able`	turn LDB write–through on or off
`ininames`	record names of initial data files for local data block
`lclear`	erase LDB entries of one scope
`ldb_init`	initialize (or reinitialize) the local data block
`lreset`	reinitialize LDB entries of one scope
`lstore`	copy everything from screen to LDB

11.8. CURSOR CONTROL

The following routines control the positioning and display of the cursor on the active screen.

`ascroll`	scroll to a given occurrence
`backtab`	backtab to the start of the last unprotected field
`c_off`	turn the cursor off
`c_on`	turn the cursor on
`c_vis`	turn cursor position display on or off
`disp_off`	get displacement of cursor from start of field
`getcurno`	get current field number
`gofield`	move the cursor into a field

`home`	home the cursor
`last`	position the cursor in the last field
`nl`	position cursor to the first unprotected field beyond the current line
`occur_no`	get the current occurrence number
`off_gofield`	move the cursor into a field, offset from the left
`rscroll`	scroll an array
`sh_off`	determine the cursor location relative to the start of a shifting field
`tab`	move the cursor to the next unprotected field

11.9. MESSAGE DISPLAY

The following routines are intended for the access and display of runtime application messages.

`d_msg_line`	display a message on the status line
`emsg`	display an error message and reset the message line, without turning on the cursor
`err_reset`	display an error message and reset the status line
`m_flush`	flush the message line
`msg`	display a message at a given column on the status line
`msg_get`	find a message given its number
`msgfind`	find a message given its number
`msgread`	read message file into memory
`mwindow`	display a status message in a window
`query_msg`	display a question, and return a yes or no answer
`qui_msg`	display a message preceded by a constant tag, and reset the message line
`quiet_err`	display error message preceded by a constant tag, and reset the status line
`setbkstat`	set background text for status line
`setstatus`	turn alternating background status message on or off

11.10. SCROLLING AND SHIFTING

The following routines provide access to shifting and scrolling fields and arrays.

`achg`	change the display attribute of an occurrence within a scrolling array
`ascroll`	scroll to a given occurrence
`doccur`	delete occurrences
`ioccur`	insert blank occurrences into an array
`max_occur`	get the maximum number of occurrences
`num_occurs`	find the highest numbered occurrence containing data
`oshift`	shift a field by a given amount
`rscroll`	scroll an array
`sc_max`	alter the maximum number of items allowed in a scrollable array
`sh_off`	determine the cursor location relative to the start of a shifting field
`t_scroll`	test whether an array can scroll
`t_shift`	test whether field can shift
`tst_all_mdts`	find first modified occurrence

11.11. MASS STORAGE AND RETRIEVAL

The following routines move data to or from sets of fields in the screen or LDB.

`rd_part`	read part of a data structure to the current screen
`rdstruct`	read data from a structure to the screen
`restore_data`	restore previously saved data to the screen
`rrecord`	read data from a structure to a data dictionary record
`wrecord`	write data from a data dictionary record to a structure
`wrt_part`	write part of the screen to a structure
`wrtstruct`	write data from the screen to a structure

11.12.
VALIDATION

The following routines provide an application interface to the field and group validation processes.

`bitop`	manipulate validation and data editing bits
`ckdigit`	validate check digit
`fval`	force field validation
`gval`	force group validation
`novalbit`	forcibly invalidate a field
`s_val`	validate the current screen

11.13.
GLOBAL DATA AND CHANGING JAM'S BEHAVIOR

The following routines grant access to global data and provide a way to manipulate certain aspects of JAM and Screen Manager behavior.

`async`	install an asynchronous function
`dd_able`	turn LDB write–through on or off
`finquire`	obtain information about a field
`gp_inquire`	obtain information about a group
`inquire`	obtain value of a global integer variable
`isabort`	test and set the abort control flag
`iset`	change value of integer global variable
`keyfilter`	control keystroke record/playback filtering
`keyoption`	set cursor control key options
`li_func`	install an application hook function
`msgread`	read message file into memory
`option`	set a Screen Manager option
`pinquire`	obtain value of a global strings
`pset`	Modify value of global strings

`resize`	dynamically change the size of the display
`uinstall`	install an application function

11.14. SOFT KEYS AND KEYSETS

The following routines provide an application interface to JAM's soft key support.

`c_keyset`	close a keyset
`keyset`	open a keyset
`kscscope`	query current keyset scope
`ksinq`	inquire about key set information
`ksoff`	turn off key labels
`kson`	turn on key labels
`skinq`	obtain soft key information by position
`skmark`	mark or unmark a softkey label by position
`skset`	set characteristics of a soft key by position
`skvinq`	obtain soft key information by value
`skvmark`	mark a soft key by value
`skvset`	set characteristics of a soft key by value

11.15. JAM EXECUTIVE CONTROL

The following routines, available only to applications using the JAM Executive, provide JAM Executive services.

`getjctrl`	get control string associated with a key
`jclose`	close current window or form under JAM Executive control
`jform`	display a screen as a form under JAM control
`jtop`	start the JAM Executive
`jwindow`	display a window at a given position under JAM control
`putjctrl`	associate a control string with a key

11.16. BLOCK MODE CONTROL

The following routines are used in applications requiring block mode support.

`blkdrvr`	install block mode driver
`blkinit`	initialize (and turn on) block mode terminal
`blkreset`	reset (and turn off) block mode terminal

11.17. MISCELLANEOUS

`fi_path`	return the full path name of a file
`jplcall`	execute a JPL procedure
`jplload`	execute the JPL load command
`jplpublic`	execute the JPL public command
`jplunload`	execute the JPL unload command
`l_close`	close a library
`l_open`	open a library
`sdtime`	get formatted system date and time
`udtime`	format user-supplied date and time

Chapter 12.
Function Reference

All JAM function names begin with the prefix `xsm_`. In the Function Reference Chapter functions are listed without the prefix and, in a few cases, under a name that is not a portion of the function name — but that is suggestive of the utility of the function. For example, the function `xsm_r_at_cur`, which displays a window at a specified position, is found under the listing name `window`, along with the function `xsm_r_window`. In these cases, the calling syntax of each function is listed under the SYNOPSIS section of the listing.

For each entry, you will find several sections:

- A synopsis similar to a COBOL function declaration, giving the types of the arguments and return value.
- A description of the function's arguments, prerequisites, results, and side–effects.
- The function's return values, if any, and their meanings.
- A list of variants.
- A list of functions that perform related tasks.
- An example illustrating the function's use.

Header files that need to be copied are indicated in the synopsis section.

To view functions by category, refer to the Library Function Overview (chapter 11.) To view a complete list of functions alphabetically by the actual function name (including the `xsm_` prefix), see the Library Function Index (chapter 13.).

achg

change the display attribute of an occurrence within a scrolling array

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 field-number      pic S(9)9 comp-5.
77 occurrence        pic S(9)9 comp-5.
77 display-attribute pic S(9)9 comp-5.
77 status            pic S(9)9 comp-5.
call "xsm_o_achg" using field-number, occurrence,
       display-attribute giving status.
```

DESCRIPTION

NOTE: This function has only two variants, `xsm_o_achg` and `xsm_i_achg`. There is NO `xsm_achg`.

This function changes the display attribute of an occurrence within a scrollable array. If the occurrence is onscreen, the attribute with which the occurrence is currently displayed is changed as well. When the occurrence is scrolled to another position within the array the new attribute moves with the occurrence. Use `xsm_chg_attr` if you want all of the occurrences within the array to scroll through an attribute so that their appearance is determined by their onscreen positions.

Possible values for the argument `display-attribute` are defined in the header file `smattrib.incl.cobol`, as shown in the table below:

Foreground Attributes	*Background Attributes*
ATT-BLANK	ATT-BHILIGHT
ATT-REVERSE	
ATT-UNDERLN	
ATT-BLINK	
ATT-HILIGHT	
ATT-STANDOUT	

Foreground Attributes	*Background Attributes*
ATT–DIM	
ATT–ACS (alternate character set)	
Foreground Colors	*Background Colors*
ATT–BLACK	ATT–BBLACK
ATT–BLUE	ATT–BBLUE
ATT–GREEN	ATT–BGREEN
ATT–CYAN	ATT–BCYAN
ATT–RED	ATT–BRED
ATT–MAGENTA	ATT–BMAGENTA
ATT–YELLOW	ATT–BYELLOW
ATT–WHITE	ATT–BWHITE

Foreground colors may be used alone or with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting the foreground value will cause the attribute to default to black.

If `display-attribute` is zero, the occurrence's display attribute is removed, leaving it with the field display attribute. Then, if that occurrence is onscreen, it is displayed with the attribute attached to its field.

This function will not work on an array that is not scrollable. Use `xsm_chg_attr` to change the display attribute of an individual field.

RETURNS

–1 if the field isn't found or isn't scrollable, or if `occurrence` is invalid. 0 otherwise.

VARIANTS

```
call "xsm_i_achg" using field-name, occurrence,
       display-attribute giving status.
```

RELATED FUNCTIONS

```
call "xsm_chg_attr" using field-number, display-attribute
       giving status.
```

allget

load screen from the LDB

SYNOPSIS

```
77 respect-flag       pic S(9)9 comp-5.
call "xsm_allget" using respect-flag.
```

DESCRIPTION

This function copies data from the local data block to fields on the current screen with matching names.

If `respect-flag` is nonzero, this function does not write to fields that already contain data, or that have their MDT bits set. If the flag is zero, all fields are initialized. When this function is called by the **JAM** run–time system, or by your screen entry function, it does *not* set MDT bits for the fields it initializes.

This function is called automatically by the **JAM** screen–display logic, unless LDB processing has been turned off using `xsm_dd_able`. Application code should not normally need to call it.

RELATED FUNCTIONS

```
call "xsm_dd_able" using flag.
call "xsm_lstore" giving status.
```

amt_format

write data to a field, applying currency editing

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 buffer             display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_amt_format" using field-number, buffer giving status.
```

DESCRIPTION

If the specified field has a currency edit, it is applied to the data in `buffer`. If the resulting string is too long for the field, an error message is displayed. Otherwise, `xsm_put=field` is called to write the edited string to the specified field.

If the field has no currency edit, `xsm_putfield` is called with the unedited string.

RETURNS

–1 if the field is not found or the occurrence is out of range;
–2 if the edited string will not fit in the field;
0 otherwise.

VARIANTS

```
call "xsm_e_amt_format" using field-name, element, buffer
       giving status.
call "xsm_i_amt_format" using field-name, occurrence, buffer
       giving status.
call "xsm_n_amt_format" using field-name, buffer giving status.
call "xsm_o_amt_format" using field-number, occurrence, buffer
       giving status.
```

RELATED FUNCTIONS

```
call "xsm_dtofield" using field-number, value, format giving
       status.
call "xsm_strip_amt_ptr" using field-number, inbuf,  giving
       outbuf.
```

ascroll

scroll to a given occurrence

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 occurrence         pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_ascroll" using field-number, occurrence giving
       status.
```

DESCRIPTION

This function scrolls the designated field so that the indicated `occurrence` appears there. Synchronized arrays will scroll along with the target array.

The field need not be the first element of a scrolling array. You can use this function, for instance, to place the nineteenth occurrence in the third onscreen element of a five–element scrolling array.

The validity of certain combinations of parameters depends on the exact nature of the field. For instance, if field number 7 is the third element of a scrolling array and `occurrence` is 1 a call to `xsm_ascroll` will fail on a non–circular scrolling array but succeed if scrolling is circular.

RETURNS

–1 if field or occurrence specification is invalid,
0 otherwise.

VARIANTS

```
call "xsm_n_ascroll" using field-name, occurrence giving
       status.
```

RELATED FUNCTIONS

```
call "xsm_rscroll" using field-number, req-scroll giving lines.
call "xsm_t_scroll" using field-number giving status.
```

async

install an asynchronous function

SYNOPSIS

```
77 func                entry.
77 timeout             pic S(9)9 comp-5.
call "xsm_async" using func, timeout.
```

DESCRIPTION

This routine installs a a function that will be called regularly during keyboard processing (ie. - `xsm_input`). The first parameter is the address of the function. Use the operating system subroutine `s$find_entry` to find the entry point. The second parameter is the timeout, in tenths of a second, between subsequent function calls.

The asynchronous function is called only when the keyboard is being read, and only if a keystroke does not arrive within the specified timeout. The authoring utility, `jxform`, uses an asynchronous function to update its cursor position display. An asynchronous function might also be used to implement a real-time clock display.

RELATED FUNCTIONS

```
call "xsm_uinstall" using  usage, func, func-name giving
       status.
```

backtab

backtab to the start of the last unprotected field

SYNOPSIS

```
call "xsm_backtab".
```

DESCRIPTION

When the cursor is in a field unprotected from tabbing into, but not in the first enterable position, it is moved to the first enterable position of that field. However, if the cursor is in a field with a previous–field edit and one of the fields specified by the edit is unprotected from tabbing, the cursor is moved to the first enterable position of that field. Otherwise, the cursor is moved to the first enterable position of the tab–unprotected field with the next lowest field number. If the cursor is in the first position of the first unprotected field on the screen, or before the first unprotected field on the screen, it wraps backward into the last unprotected field. When there are no unprotected fields, the cursor doesn't move.

If the destination field is shiftable, it is reset according to its justification. The first enterable position depends on the justification of the field and, in fields with embedded punctuation, on the presence of punctuation.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

This function is called when the JAM logical key BACK is struck.

RELATED FUNCTIONS

```
call "xsm_home" giving field-number.
call "xsm_last".
call "xsm_nl".
call "xsm_tab".
```

base_fldno

get the field number of the first element of an array

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 base-number        pic S(9)9 comp-5.
call "xsm_base_fldno" using field-number giving base-number.
```

DESCRIPTION

A base field number is the field number of the first element of an array. Use `xsm_base_fldno` to obtain the base field number of an array.

RETURNS

The field number of the base element of the array containing the specified field, or 0 if the field number is out of range.

bel

beep!

SYNOPSIS

```
call "xsm_bel".
```

DESCRIPTION

Causes the terminal to beep, ordinarily by transmitting the ASCII BEL code to it. If there is a BELL entry in the video file, xsm_bel will transmit that instead, usually causing the terminal to flash instead of beeping.

Even if there is no BELL entry, use this function instead of sending a BEL, because certain displays use BEL as a graphics character.

Including a **%B** at the beginning of a message displayed on the status line will cause this function to be called.

bitop

manipulate validation and data editing bits

SYNOPSIS

```
copy "smbitops.incl.cobol".

77 field-number       pic S(9)9 comp-5.
77 action             pic S(9)9 comp-5.
77 bit                pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_bitop" using field-number, action, bit giving status.
```

DESCRIPTION

You can use this function to inspect and modify validation and data editing bits of screen fields, without reference to internal data structures. The first parameter identifies the field to be operated upon.

`action` can include a test and at most one manipulation from the following table of values, which are defined in `smbitops.incl.cobol`:

Value	*Meaning*
BIT-CLR	Turn bit off
BIT-SET	Turn bit on
BIT-TOGL	Flip state of bit
BIT-TST	Report state of bit

The third parameter is a bit identifier, drawn from the following table:

Character edits				
N-ALL	N-DIGIT	N-YES-NO	N-ALPHA	N-NUMERIC
N-ALPHNUM	N-FCMASK			

Field edits	*Field edits*			
N-RTJUST	N-REQD	N-VALIDED	N-MDT	N-CLRINP
N-MENU	N-UPPER	N-LOWER	N-RETENTRY	N-FILLED
N-NOTAB	N-WRAP	N-ADDLEDS	N-EPROTECT	N-TPROTECT
N-CPROTECT	N-VPROTECT	N-ALLPROTECT	N-SELECTED	

The character edits are not, strictly speaking, bits; you cannot toggle them, but the other functions work as you would expect. N-ALLPROTECT is a special value meaning all four protect bits at once.

N-VALIDED and N-MDT are the only bit operations that can apply to individual offscreen and onscreen occurrences. The protection operations can apply to an array as a whole, including offscreen occurrences (see xsm_aprotect). All other bit operations are attached to fixed onscreen positions.

The variants xsm_e_bitop and xsm_n_bitop can take a group name as an argument. The function will then affect the group bits.

This function has two additional variants, xsm_a_bitop and xsm_t_bitop, which perform the requested bit operation on all elements of an array. Their synopsis appear below. If you include BIT-TST, these variants return 1 only if bit is set for *every* element of the array. The variants xsm_i_bitop and xsm_o_bitop are restricted to N-VALIDED and N-MDT.

RETURNS

1 if there was no error, the action included
–1 if the field or occurrence cannot be found
–2 if the action or bit identifiers are invalid; a test operation, and bit was set
–3 if xsm_i_bitop or xsm_o_bitop was called with bit set to something other than N-VALIDED or N-MDT
0 otherwise.

VARIANTS

```
call "xsm_a_bitop" using array-name, action, bit giving status.
call "xsm_e_bitop" using array-name, element, action, bit
       giving status.
call "xsm_i_bitop" using array-name, occurrence, action, bit
       giving status.
call "xsm_n_bitop" using name, action, bit giving status.
```

```
call "xsm_o_bitop" using field-number, occurrence, action, bit
      giving status.
call "xsm_t_bitop" using array-number, action, bit giving
      status.
```

bkrect

set background color of rectangle

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 start-line         pic S(9)9 comp-5.
77 start-column       pic S(9)9 comp-5.
77 num-of-lines       pic S(9)9 comp-5.
77 number-of-columns  pic S(9)9 comp-5.
77 background-colors  pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_bkrect" using start-line, start-column, num-of-lines,
      number-of-columns, background-colors giving status.
```

DESCRIPTION

This function changes the background color of a rectangular area of the current screen. Any fields or elements that begin within the rectangular area will have their background attributes changed to the specified attribute. This means that if there are any fields or elements that are not entirely contained within the rectangular area, a ragged edge will result. Display text that falls with in the rectangular area will have its background attribute set.

The arguments `start-line` and `start-column` can have any value from 1 through the number of lines (or columns) on the screen.

The background color must be one of the values defined in `smattrib.incl.cobol` (ATT-BBLACK, ATT-BBLUE, etc.). You can highlight the background color by summing the background color attribute with ATT-BHILIGHT.

RETURNS

-1 if the starting line or column was invalid.
1 if the starting line and column were valid, but the rectangle had to be truncated to fit.
0 if no error.

blkinit

initialize (and turn on) block mode terminal

SYNOPSIS

```
77 return-value       pic S(9)9 comp-5.
call "xsm_blkinit" giving return-value.
```

DESCRIPTION

This routine must be called by the application program to initiate block mode terminal action. A block mode terminal driver must have been previously installed.

This routine checks that a block mode terminal driver is installed. If a driver is found, it is called. The driver should return 0 if all is successful.

Generally the return code can be ignored. If the terminal cannot be put into block mode it will still work (possibly better) in interactive mode.

If the driver signifies that all is OK, the global variable `sm_blkcontrol` is set to point to the local block terminal control handler. All Screen Manager calls for block mode support are made through this control routine.

On the first call to the present routine the driver is called with BLK_INIT to perform any required initialization.

On subsequent calls BLK_BLOCK is called instead of BLK_INIT.

RETURNS

return value from driver if one exists.
-1 otherwise.

RELATED FUNCTIONS

```
call "xsm_blkreset" giving return-value.
```

blkreset

reset (and turn off) block mode terminal

SYNOPSIS

```
77 return-value       pic S(9)9 comp-5.
call "xsm_blkreset" giving return-value.
```

DESCRIPTION

This routine must be called by the application program to reset block mode terminal action. A block mode terminal driver must have been previously installed.

This routine checks that a block mode terminal driver is installed. If a driver is found, it is called. The driver should return 0 if all is successful.

Generally the return code can be ignored as the terminal is often already in interactive mode. The exception is on those systems that are normally block mode. Many JAM programs rely on the fact that the terminal can be put into interactive mode.

Note that the driver is called with BLK_CHAR, not with BLK_RESET. The only time the driver is called for a full reset is when JAM is about to go to the operating system – either exiting or performing a "shell escape".

RETURNS

return value from driver if one exists.
–1 otherwise.

RELATED FUNCTIONS

```
call "xsm_blkinit" giving return-value.
```

c_keyset

close a keyset

SYNOPSIS

```
copy "smsoftk.incl.cobol".

77 scope              pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_c_keyset" using scope giving status.
```

DESCRIPTION

This function closes the keyset of the given scope. It frees all memory associated with the keyset and marks that scope as free. If the keyset was currently displayed, the keyset labels are changed to reflect the new keyset.

See the keyset chapter of the Author's Guide for a detailed explanation of keyset scopes.

Scope Value from `smsoftk.incl.cobol`	*Description*
KS-APPLIC	Application scope.
KS-FORM	Form or window scope.
KS-SYSTEM	`jxform` system key sets.

Use `xsm_d_keyset` and `xsm_r_keyset` to open keysets.

RETURNS

0 if there is no error
-2 if there is no keyset currently at that scope
-3 if the scope is out of range

RELATED FUNCTIONS

```
call "xsm_r_keyset" using name, scope giving status.
call "xsm_d_keyset" using ADDRESS, scope giving status.
```

c_off

turn the cursor off

SYNOPSIS

```
call "xsm_c_off".
```

DESCRIPTION

This function notifies **JAM** that the normal cursor setting is off. The normal setting is in effect except:

- When a block cursor is in use, as during menu processing, the cursor is off.
- While Screen Manager functions are writing to the display the cursor is off.
- Within certain error message display functions the cursor is on.

If the display cannot turn its cursor on and off (V-CON and V-COF entries are not defined in the video file), this function will have no effect.

Use `xsm_c_on` to turn the cursor on.

RELATED FUNCTIONS

```
call "xsm_c_on".
```

c_on

turn the cursor on

SYNOPSIS

```
call "xsm_c_on".
```

DESCRIPTION

This function notifies JAM that the normal cursor setting is on. The normal setting is in effect except:

- When a block cursor is in use, as during menu processing, the cursor is off.
- While Screen Manager functions are writing to the display the cursor is off.
- Within certain error message display functions the cursor is on.

If the display cannot turn its cursor on and off (V-CON and V-COF entries are not defined in the video file), this function will have no effect.

Use `xsm_c_off` to turn the cursor off.

RELATED FUNCTIONS

```
call "xsm_c_off".
```

c_vis

turn cursor position display on or off

SYNOPSIS

```
77 display            pic S(9)9 comp-5.
call "xsm_c_vis" using display.
```

DESCRIPTION

Assigning a non–zero value to `display` displays subsequent status line messages with the cursor's position display. This includes background status messages. Messages that would overlap the cursor position display are truncated.

Setting `display` to zero will cause subsequent status line messages to be displayed without the cursor's position display.

This function will have no effect if the `CURPOS` entry in the video file is not defined. In that case the cursor position display will never appear.

JAM uses an asynchronous function and a status line function to perform the cursor position display. If the application has previously installed either of those, this function will override it.

calc

execute a math edit style expression

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 occurrence         pic S(9)9 comp-5.
77 expression         display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_calc" using field-number, occurrence, expression
       giving status.
```

DESCRIPTION

Use `xsm_calc` to execute a math edit style expression. With this function you can perform mathematical operations that use the contents of one or more fields and then insert the result into a field.

The third parameter `expression` is a math edit style expression. See the JAM Author's Guide for a complete description on how to create the expression.

The first two parameters, `field-number` and `occurrence` identify the field and occurrence with which the calculation is associated. Normally you will not need to use them and should set them both to 0.

If you want to use relative references to fields in your expression, use the arguments `field-number` and `occurrence` to specify the field to which they should be relative.

If in the event of a math error you want the cursor to move a specific field, specify that field with `field-number`. In addition, if the desired field is an occurrence within an array, specifying the `occurrence` will cause the referenced array to scroll to `field-number`.

RETURNS

–1 is returned if a math error occurred.
0 is returned otherwise.

cancel

reset the display and exit

SYNOPSIS

```
77 arg                pic S(9)9 comp-5.
call "xsm_cancel" using arg.
```

DESCRIPTION

This function is installed by `xsm_initcrt` to be executed if a keyboard interrupt occurs. It calls `xsm_resetcrt` to restore the display to the operating system's default state, and exits to the operating system.

If your operating system supports it, you can also install this function to handle conditions that normally cause a program to abort. If a program aborts without calling `xsm_resetcrt`, you may find your terminal in an odd state; `xsm_cancel` can prevent that.

The argument `arg` is a dummy argument. It should have the value zero.

chg_attr

change the display attribute of a field

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 field-number       pic S(9)9 comp-5.
77 display-attribute  pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_chg_attr" using field-number, display-attribute
       giving status.
```

DESCRIPTION

Use this function to change the display attribute of an individual field or an element within an array. To change an occurrence attribute so that the attribute moves with the occurrence use `xsm_o_achg`.

If the field is part of a scrolling array, then each occurrence may also have a display attribute that overrides the field display attribute when the occurrence arrives onto the screen.

Possible values for `display-attribute` are defined in `smattrib.incl.cobol`, as shown in the table below:

Foreground Attributes	*Background Attributes*
ATT–BLANK	ATT–BHILIGHT
ATT–REVERSE	
ATT–UNDERLN	
ATT–BLINK	
ATT–HILIGHT	
ATT–STANDOUT	
ATT–DIM	
ATT–ACS (alternate character set)	

Foreground Colors	*Background Colors*
ATT-BLACK	ATT-BBLACK
ATT-BLUE	ATT-BBLUE
ATT-GREEN	ATT-BGREEN
ATT-CYAN	ATT-BCYAN
ATT-RED	ATT-BRED
ATT-MAGENTA	ATT-BMAGENTA
ATT-YELLOW	ATT-BYELLOW
ATT-WHITE	ATT-BWHITE

Foreground colors may be used alone or added together with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting the foreground value will cause the attribute to default to black.

NOTE: The variant `xsm_o_chg_attr` does not take the usual arguments. The second argument is an element rather than an occurrence.

RETURNS

-1 if the field is not found
0 otherwise.

VARIANTS

```
call "xsm_e_chg_attr" using field-name, element,
      display-attribute giving status.
call "xsm_n_chg_attr" using field-name, display-attribute
      giving status.
call "xsm_o_chg_attr" using field-number, element,
      display-attribute giving status.
```

RELATED FUNCTIONS

```
call "xsm_o_achg" using field-number, occurrence,
      display-attribute giving status.
```

ckdigit

validate check digit

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 field-data         display-2 pic x(256).
77 occurrence         pic S(9)9 comp-5.
77 modulus            pic S(9)9 comp-5.
77 minimum-digits     pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_ckdigit" using field-number, field-data, occurrence,
       modulus, minimum-digits giving status.
```

DESCRIPTION

This function is called by field validation. It verifies that `field-data` contains the required minimum number of digits terminated by the proper check digit. If not, it posts an error message before returning. It can also be used to check any character string or field. If `field-data` is null, the string to check is obtained from the `field-number` and `occurrence` and an error message is displayed if the string is bad. If `field-number` is zero, no message will be posted, but the function's return code will indicate whether the string passed its check.

A fuller description of `sm_ckdigit` is included with the source code, which is distributed with **JAM**.

Note that this function can be replaced by a user-installed check digit function which field validation will call instead. See the chapter on installing functions.

RETURNS

0 If the field contents are available and valid.
-1 If the field contents do not contain the minimum number of digits or the proper check digit.
-2 If the length of `field-data` is zero and the field or occurrence cannot be found

cl_all_mdts

clear all MDT bits

SYNOPSIS

```
call "xsm_cl_all_mdts".
```

DESCRIPTION

Clears the MDT (modified data tag) of every occurrence, both onscreen and off.

JAM sets the MDT bit of an occurrence to indicate that it has been modified, either by keyboard entry or by a call to a function like `xsm_putfield`, since the screen was first displayed (i.e., after the screen entry function returns).

RELATED FUNCTIONS

```
call "xsm_tst_all_mdts" using occurrence giving field-number.
```

cl_unprot

clear all unprotected fields

SYNOPSIS

```
call "xsm_cl_unprot".
```

DESCRIPTION

Erases onscreen and offscreen data from all fields that are not protected from clearing (CPROTECT). Date and time fields that take system values are re–initialized. Fields with the null edit are reset to their null indicator values.

This function is normally bound to the CLEAR ALL key.

RELATED FUNCTIONS

```
call "xsm_aprotect" using field-number, mask giving status.
```

clear_array

clear all data in an array

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_clear_array" using field-number giving status.

call "xsm_1clear_array" using field-number giving status.
```

DESCRIPTION

Both functions clear all data from the array containing the field specified by field–number. The value returned by `xsm_num_occurs` is changed to zero. The array is cleared even if it is protected from clearing (CPROTECT).

`xsm_clear_array` also clears arrays synchronized with the specified array, except for synchronized arrays that are protected from clearing.

`xsm_1clear_array` only clears the specified array.

RETURNS

–1 if the field does not exist;
0 otherwise.

VARIANTS

```
call "xsm_n_clear_array" using field-name giving status.
call "xsm_n_1clear_array" using field-name giving status.
```

RELATED FUNCTIONS

```
call "xsm_aprotect" using field-number, mask giving status.
call "xsm_protect" using field-number giving status.
```

close_window

close current window

SYNOPSIS

```
77                      pic S(9)9 comp-5.
call "xsm_close_window" giving status.
```

DESCRIPTION

`xsm_close_window` is used to close a window opened by `xsm_r_window` (or variant), `xsm_r_at_cur` (or variant), or `xsm_mwindow`.

The currently open window is erased, and the screen is restored to the state before the window was opened. All data from the window being closed is lost unless LDB processing is active, in which case named fields are copied to the LDB using `xsm_lstore`. Since windows are stacked, the effect of closing a window is to return to the previous window. The cursor reappears at the position it had before the window was opened.

When using the **JAM** Executive, use `xsm_jclose` to close a form. `xsm_jclose` will call `xsm_jform` to pop the form stack and open the new top form on the stack. In the case of a window, `xsm_jclose` will call `xsm_close_window` to close the window.

RETURNS

-1 is returned if there is no window open, (i.e. if the currently displayed screen is a form or if no screen is displayed).
0 is returned otherwise.

RELATED FUNCTIONS

```
call "xsm_r_window" using screen-name, start-line, start-column
       giving status.
call "xsm_wselect" using window-number giving return-value.
```

d_msg_line

display a message on the status line

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 message           display-2 pic x(256).
77 display-attribute pic S(9)9 comp-5.
call "xsm_d_msg_line" using message, display-attribute.
```

DESCRIPTION

The message in `message` is displayed on the status line, with an initial display attribute of `display-attribute`. If the cursor position display has been turned on (see `xsm_c_vis`), the end of the status line will contain the cursor's current row and column. Messages displayed with `xsm_d_msg_line` override both background and field status text.

Messages posted with `xsm_d_msg_line` are displayed until the status line is cleared by `xsm_d_msg_line`. They will persist from screen to screen until cleared. Clearing is accomplished by passing `xsm_d_msg_line` an empty string for `message` and a 0 for `display-attribute`. Once cleared, any currently overidden message will resume. The function `xsm_d_msg_line` will itself be overridden by `xsm_err_reset` and related functions, or by the ready/wait message enabled by `xsm_setstatus`.

Possible values for `display-attribute` are defined in `smattrib.incl.cobol`, as shown in the table below:

Attribute Value	*Hex Code*	*Attribute Value*	*Hex Code*
Foreground Highlights		Background Highlights	
ATT-BLANK	0008	ATT-BHILIGHT	8000
ATT-REVERSE	0010		
ATT-UNDERLN	0020		
ATT-BLINK	0040		
ATT-HILIGHT	0080		
ATT-STANDOUT	0800		
ATT-DIM	1000		
ATT-ACS (alternate character set)	2000		
Foreground Colors		Background Colors	
ATT-BLACK	0000	ATT-BBLACK	0000
ATT-BLUE	0001	ATT-BBLUE	0100
ATT-GREEN	0002	ATT-BGREEN	0200
ATT-CYAN	0003	ATT-BCYAN	0300
ATT-RED	0004	ATT-BRED	0400
ATT-MAGENTA	0005	ATT-BMAGENTA	0500
ATT-YELLOW	0006	ATT-BYELLOW	0600
ATT-WHITE	0007	ATT-BWHITE	0700

Foreground colors may be used alone or added together with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting the foreground value will cause the attribute to default to black.

Several *percent escapes* provide control over the content and presentation of status messages. The character following the percent sign must be in upper-case. Note that, if a message containing percent escapes is displayed before `xsm_initcrt` is called, the percent escapes will show up in the message.

If a string of the form `%A`nnnn appears anywhere in the message, the hexadecimal number nnnn is interpreted as a display attribute to be applied to the remainder of the message. The

table gives the numeric values of the logical display attributes you will need to construct embedded attributes. If you want a digit to appear *immediately* after the attribute change, pad the attribute to 4 digits with leading zeros. If the following character is not a legal hex digit, then leading zeros are unnecessary.

If a string of the form `%K`keyname appears anywhere in the message, keyname is interpreted as a logical key value, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the `%K` is stripped out and the value remains. Key values are defined in `smkeys.incl.cobol`; it is of course the name, not the number, that you want here. The value must be in upper-case.

If the message begins with a `%B`, **JAM** will beep the terminal (using `xsm_bel`) before issuing the message.

RELATED FUNCTIONS

```
call "xsm_err_reset" using message.
call "xsm_msg" using column, disp-length, text.
call "xsm_mwindow" using text, line, column giving status.
```

dblval

get the value of a field as a real number

SYNOPSIS

```
77 value              pic S9(9) comp-2.
77 field-number       pic S(9)9 comp-5.
call "xsm_dblval" using field-number giving value.
```

DESCRIPTION

This function returns the contents of `field-number` as a real number. It calls `xsm_strip_amt_ptr` to remove superfluous amount editing characters before converting the data.

RETURNS

The real value of the field is returned.
If the field is not found, the function returns 0.

VARIANTS

```
call "xsm_e_dblval" using field-name, element giving value.
call "xsm_i_dblval" using field-name, occurrence giving value.
call "xsm_n_dblval" using field-name giving value.
call "xsm_o_dblval" using field-number, occurrence giving
      value.
```

RELATED FUNCTIONS

```
call "xsm_dtofield" using field-number, value, format giving
      status.
call "xsm_strip_amt_ptr" using field-number, inbuf,  giving
      outbuf.
```

dd_able

turn LDB write–through on or off

SYNOPSIS

```
77 flag                pic S(9)9 comp-5.
call "xsm_dd_able" using flag.
```

DESCRIPTION

During normal **JAM** processing, named fields in the screen and local data block are kept in sync. When a screen is displayed (and after the screen entry function completes), values are copied in from the LDB; when control passes from the screen (before the screen entry function is executed), values are copied back to the LDB. Normally, when application code reads or writes a value to or from a named field/LDB entry **JAM** treats the name as a field name unless no such field exists, in which case **JAM** treats the name as an LDB entry name. During screen entry and exit processing, this logic is reversed in order to preserve the illusion that screen and LDB entries that share the same name also share the same data.

`xsm_dd_able` turns this feature off if `flag` is "0" and on if it is "1". The feature is on by default. When it is off, the LDB is never accessed.

deselect

deselect a checklist occurrence

SYNOPSIS

```
77 group-name          display-2 pic x(256).
77 group-occurrence    pic S(9)9 comp-5.
77 status              pic S(9)9 comp-5.
call "xsm_deselect" using group-name, group-occurrence giving
       status.
```

DESCRIPTION

This function allows you to deselect a specific occurrence within a checklist. The group name and occurrence number is used to reference the desired selection. See the Author's Guide for a more detailed discussion of groups.

Use `xsm_select` to select a group occurrence and `xsm_isselected` to check whether or not a particular group occurrence is currently selected.

NOTE: You can not deselect a radio button occurrence. Using `xsm_select` on a radio button occurrence will automatically deselect the current selection.

RETURNS

-1 arguments do not reference a checklist occurrence.
0 occurrence not previously selected.
1 occurrence previously selected.

RELATED FUNCTIONS

```
call "xsm_isselected" using group-name, group-occurrence giving
       status.
call "xsm_select" using group-name, group-occurrence giving
       status.
```

dicname

set data dictionary name

SYNOPSIS

```
77 dic-name          display-2 pic x(256).
77 status            pic S(9)9 comp-5.
call "xsm_dicname" using dic-name giving status.
```

DESCRIPTION

This function names the application's data dictionary, which is *data.dic* by default. It must be called before **JAM** initialization, in particular before `xsm_ldb_init` is called to initialize the local data block from the data dictionary. The argument `dic-name` is a character string giving the file name; **JAM** will search for it in all the directories in the `SMPATH` variable.

You can achieve the same effect by defining the `SMDICNAME` variable in your setup file equal to the data dictionary name. See the section on setup files in the Configuration Guide.

Use the function `xsm_pinquire` to find the name of the data dictionary in use.

RETURNS

–1 if it fails to allocate memory to store the name,
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_pinquire" using which giving buffer.
```

disp_off

get displacement of cursor from start of field

SYNOPSIS

```
77 offset              pic S(9)9 comp-5.
call "xsm_disp_off" giving offset.
```

DESCRIPTION

Returns the difference between the first column of the current field and the current cursor location. This function ignores offscreen data; use `xsm_sh_off` to obtain the total cursor offset of a shiftable field.

RETURNS

The difference between cursor position and start of field, or
-1 if the cursor is not in a field.

RELATED FUNCTIONS

```
call "xsm_getcurno" giving field-number.
call "xsm_sh_off" giving offset.
```

dlength

get the length of a field's contents

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 data-length        pic S(9)9 comp-5.
call "xsm_dlength" using field-number giving data-length.
```

DESCRIPTION

Returns the length of data stored in `field-number`. The length does not include leading blanks in right justified fields, or trailing blanks in left–justified fields (which are also ignored by `xsm_getfield`). It does include data that have been shifted offscreen.

RETURNS

Length of field contents, or
–1 if the field is not found.

VARIANTS

```
call "xsm_e_dlength" using field-name, element giving
      data-length.
call "xsm_i_dlength" using field-name, occurrence giving
      data-length.
call "xsm_n_dlength" using field-name giving data-length.
call "xsm_o_dlength" using field-number, occurrence giving
      data-length.
```

RELATED FUNCTIONS

```
call "xsm_length" using field-number giving field-length.
```

do_region

rewrite part or all of a screen line

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 line              pic S(9)9 comp-5.
77 column            pic S(9)9 comp-5.
77 length            pic S(9)9 comp-5.
77 display-attribute pic S(9)9 comp-5.
77 text              display-2 pic x(256).
call "xsm_do_region" using line, column, length,
      display-attribute, text.
```

DESCRIPTION

The screen region defined by `line`, `column`, and `length` is rewritten. `Line` and `column` are counted *from zero*, with (0, 0) the upper left-hand corner of the screen.

If `text` is zero, the screen region is redrawn with whatever `display-attribute` has been assigned. If `text` is shorter than `length`, it is padded out with blanks. In either case, the display attribute of the whole area is changed to `display-attribute`.

Possible values for `display-attribute` are defined in `smattrib.incl.cobol`, as shown in the table below:

Foreground Attributes	*Background Attributes*
ATT–BLANK	ATT–BHILIGHT
ATT–REVERSE	
ATT–UNDERLN	
ATT–BLINK	
ATT–HILIGHT	
ATT–STANDOUT	
ATT–DIM	
ATT–ACS (alternate character set)	

Foreground Colors	*Background Colors*
ATT–BLACK	ATT–BBLACK
ATT–BLUE	ATT–BBLUE
ATT–GREEN	ATT–BGREEN
ATT–CYAN	ATT–BCYAN
ATT–RED	ATT–BRED
ATT–MAGENTA	ATT–BMAGENTA
ATT–YELLOW	ATT–BYELLOW
ATT–WHITE	ATT–BWHITE

Foreground colors may be used alone or added together with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting the foreground value will cause the attribute to default to black.

doccur

delete occurrences

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 occurrence         pic S(9)9 comp-5.
77 count              pic S(9)9 comp-5.
77 return-value       pic S(9)9 comp-5.
call "xsm_o_doccur" using field-number, occurrence, count
       giving return-value.
```

DESCRIPTION

NOTE: This function only exists in the `o_` and `i_` variations. There is NO `xsm_doccur` since this function only applies to arrays.

This function deletes the data in `count` occurrences beginning with the specified `occurrence`. If the array is scrollable, then it deallocates `count` occurrences. The data in occurrences following the last deleted occurrence are moved up in the array so that there are no gaps. Fewer than `count` occurrences will be deleted if the number of remaining allocated occurrences, starting with the referenced occurrence, is less than `count`.

If `count` is negative, occurrences are inserted instead, subject to limitations explained at `xsm_ioccur`. The function `xsm_ioccur` is normally used to add blank occurrences.

If `occurrence` is zero, the occurrence used is that of `field-number`. If `occurrence` is nonzero, however, it is taken relative to the first field of the array in which `field-number` occurs.

Any clearing–unprotected synchronized arrays will have the same operations performed on them as the referenced array.

This function is normally bound to the DELETE LINE key.

RETURNS

–1 if the field or occurrence number was out of range;
–3 if insufficient memory was available;
otherwise, the number of occurrences actually deleted (zero or more).

VARIANTS

```
call "xsm_i_doccur" using field-name, occurrence, count giving
       return-value.
```

dtofield

write a real number to a field

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 value              pic S9(9) comp-2.
77 format             display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_dtofield" using field-number, value, format giving
       status.
```

DESCRIPTION

The real number `value` is converted to human–readable form, according to `format`, and moved into `field-number` via a call to `xsm_amt_format`. If the `format` string is empty, the number of decimal places will be taken from a data type edit, if one exists; failing that, from a currency edit, if one exists; or failing that, will default to 2.

The number of decimal places may be forced to be an arbitrary number `n`, via rounding, by using the format string `%.nf`". The format string `%t.nf`" may be used to truncate instead of to round.

RETURNS

–1 is returned if the field is not found.
–2 is returned if the output would be too wide for the destination field.
0 is returned otherwise.

VARIANTS

```
call "xsm_e_dtofield" using field-name, element, value, format
       giving status.
call "xsm_i_dtofield" using field-name, occurrence, value,
       format giving status.
call "xsm_n_dtofield" using field-name, value, format giving
       status.
call "xsm_o_dtofield" using field-number, occurrence, value,
       format giving status.
```

RELATED FUNCTIONS

```
call "xsm_amt_format" using field-number, buffer giving status.
call "xsm_dblval" using field-number giving value.
```

e_
variants that take a field name and element number

SYNOPSIS

```
77 field-name          display-2 pic x(256).
77 element             pic S(9)9 comp-5.
call "xsm_e_..." using field-name, element, ....
```

DESCRIPTION

The e_ variant functions access one element of an array by field name and element number. For a description of any particular function, look under the related function without e_ in its name. For example, xsm_e_amt_format is described under xsm_amt_format.

Despite the fact that they take a field name as argument, these functions do not search the LDB for names not found in the screen because an element number is ambiguous when referring to the LDB.

edit_ptr

get special edit string

SYNOPSIS

```
copy "smedits.incl.cobol".

77 buffer              display-2 pic x(256).
77 field-number        pic S(9)9 comp-5.
77 edit-type           pic S(9)9 comp-5.
call "xsm_edit_ptr" using field-number, edit-type giving
      buffer.
```

DESCRIPTION

This function searches the special edits area of a field or group for an edit of type `edit-type`. The `edit-type` should be one of the following values, which are defined in `smedits.incl.cobol`:

NOTE: Each of the values listed in the table below should have the suffix `-EDIT-CMD` added to it. So, for example, the value `NAMED` becomes `NAMED-EDIT-CMD`.

Edit type (add: -EDIT-CMD)	*Contents of edit string*
NAMED	Field name
CPROG	Name of field validation function
FE-CPROG	Name of field entry function
FX-CPROG	Name of field exit function
HELPSCR	Name of help screen
HARDHLP	Name of automatic help screen
HARDITM	Name of automatic item selection screen
ITEMSCR	Name of item selection screen
SUBMENU	Name of pull–down menu screen

Edit type (add: -EDIT-CMD)	*Contents of edit string*
TABLOOK	Name of screen for table–lookup validation
NEXTFLD	Next field (contains both primary and alternate fields)
PREVFLD	Previous field (contains both primary and alternate fields)
TEXT	Status line prompt
MEMO1 ... MEMO9	Nine arbitrary user–supplied text strings
JPLTEXT	Attached JPL code
CALC	Math expression executed at field exit
CKDIGIT	Flag and parameters for check digit
FTYPE	Data type for inclusion in structure
RETCODE	Return value for menu or return entry field
CMASK	Regular expression for field validation
CCMASK	Regular expression for character validation
CKBOX	Offset and attribute of checkbox in a group
ALTSC-CPROG	Name of alternate scrolling function
KEYSET	Name of keyset associated with screen.
SDATETIME	Date/time field with user format, initialized with system values.
UDATETIME	Date/time field with user format, initialized by the user.
CURRED	Currency field format, see `smedits.incl.cobol` for details.
NULLFIELD	Null field representation.
RANGEL	Low bound on range; up to 9 permitted

Edit type (add: -EDIT-CMD)	*Contents of edit string*
RANGEH	High bound on range; up to 9 permitted
EDT-BITS	Normally for internal use (see `smedits.incl.cobol` for more information.)

The string returned by `xsm_edit_ptr` contains:

- The total length of the string (including the two overhead bytes and any terminators) in its first byte.
- The edit–type code in its second byte.
- The body of the edit in the subsequent bytes. Refer to the source listing for the file `smedits.incl.cobol` for specific information on how to interpret each individual edit.

If the field has no edit of type `edit-type`, the returned buffer will contain a zero. If a field has multiple edits of one type, such as RANGEH or RANGEL, then each additional edit is added onto the end of the string following the same pattern as the first one. For example, the first byte would contain the length of the string up to the end of the body of the edit of RANGEH. Adding one to this number would give you the byte that contains the length of the string containing information on RANGEL and so forth.

This function is especially useful for retrieving user–defined information contained in MEMO edits.

In the case of groups, the edits `PREVFLD-EDIT-CMD`, `NEXTFLD-EDIT-CMD`, `CPROG-EDIT-CMD`, `FE-CPROG-EDIT-CMD`, and `FE-CPROG-EDIT-CMD` may be used to obtain group information.

RETURNS

The first (length) byte of the special edit of the field.
0 if the field or edit is not found.

VARIANTS

```
call "xsm_n_edit_ptr" using field-name, edit-type giving
      buffer.
```

emsg

display an error message and reset the message line without turning on the cursor

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 message            display-2 pic x(256).
call "xsm_emsg" using message.
```

DESCRIPTION

This function displays `message` on the status line, if it fits, or in a window if it is too long. If the cursor position display has been turned on (see `xsm_c_vis`), the end of the status line will contain the cursor's current row and column. If the message text would overlap that area of the status line, it will be displayed in a window instead. The message remains visible until the operator presses a key. The function's exact behavior in dismissing the message is subject to the error message options; see `xsm_option`.

`xsm_emsg` is identical to `xsm_err_reset`, except that it does not attempt to turn the cursor on before displaying the message. It is similar to `xsm_qui_msg`, which inserts a constant string (normally "ERROR:") before the message.

Several *percent escapes* provide control over the content and presentation of status messages. The character following the percent sign must be in upper–case. Note that, if a message containing percent escapes is displayed before `xsm_initcrt` is called, the percent escapes will show up in the message.

If a string of the form `%A`nnnn appears anywhere in the message, the hexadecimal number nnnn is interpreted as a display attribute to be applied to the remainder of the message. The table gives the numeric values of the logical display attributes you will need to construct embedded attributes. If you want a digit to appear *immediately* after the attribute change, pad the attribute to 4 digits with leading zeros. If the following character is not a legal hex digit, then leading zeros are unnecessary.

If a string of the form `%K`keyname appears anywhere in the message, keyname is interpreted as a logical key value, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the `%K` is stripped out and the value remains. Key values are defined in `smkeys.incl.cobol`; it is of course the name, not the number, that you want here. The value must be in upper–case.

If the message begins with a `%B`, JAM will beep the terminal (using `xsm_bel`) before issuing the message.

If %N appears anywhere in the message, the latter will be presented in a pop–up window rather than on the status line, and all occurrences of %N will be replaced by new lines.

If the message begins with %W, it will be presented in a pop–up window instead of on the status line. The window will appear near the bottom center of the screen, unless it would obscure the current field by so doing; in that case, it will appear near the top.

If the message begins with %Mu or %Md, **JAM** will ignore the default error message acknowledgement flag and process (for %Mu) or discard (for %Md) the next character typed.

Possible hex values for display attribute are defined in `smattrib.incl.cobol`, as shown in the table below:

Attribute Value	*Hex Code*	*Attribute Value*	*Hex Code*
Foreground Highlights		Background Highlights	
ATT–BLANK	0008	ATT–BHILIGHT	8000
ATT–REVERSE	0010		
ATT–UNDERLN	0020		
ATT–BLINK	0040		
ATT–HILIGHT	0080		
ATT–STANDOUT	0800		
ATT–DIM	1000		
ATT–ACS (alternate character set)	2000		

Attribute Value	*Hex Code*	*Attribute Value*	*Hex Code*
Foreground Colors		Background Colors	
ATT-BLACK	0000	ATT-BBLACK	0000
ATT-BLUE	0001	ATT-BBLUE	0100
ATT-GREEN	0002	ATT-BGREEN	0200
ATT-CYAN	0003	ATT-BCYAN	0300
ATT-RED	0004	ATT-BRED	0400
ATT-MAGENTA	0005	ATT-BMAGENTA	0500
ATT-YELLOW	0006	ATT-BYELLOW	0600
ATT-WHITE	0007	ATT-BWHITE	0700

Foreground colors may be used alone or added together with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting the foreground value will cause the attribute to default to black.

RELATED FUNCTIONS

```
call "xsm_err_reset" using message.
call "xsm_qui_msg" using message.
call "xsm_quiet_err" using message.
```

err_reset

display an error message and reset the status line

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 message            display-2 pic x(256).
call "xsm_err_reset" using message.
```

DESCRIPTION

The `message` is displayed on the status line until acknowledged it by pressing a key. If `message` is too long to fit on the status line, it is displayed in a window instead. If the cursor position display has been turned on (see `xsm_c_vis`), the end of the status line will contain the cursor's current row and column. If the message text would overlap that area of the status line, it will be displayed in a window instead. The exact behavior of error message acknowledgement is governed by `xsm_option`. The initial message attribute is set by `xsm_option`, and defaults to blinking.

This function turns the cursor on before displaying the message, and forces off the global flag `sm-do-not-display`. It is similar to `xsm_emsg`, which does not turn on the cursor, and to `xsm_quiet_err`, which inserts a constant string (normally "ERROR:") before the message.

Several *percent escapes* provide control over the content and presentation of status messages. See `xsm_emsg` for details.

RELATED FUNCTIONS

```
call "xsm_emsg" using message.
call "xsm_qui_msg" using message.
call "xsm_quiet_err" using message.
```

fi_path

return the full path name of a file

SYNOPSIS

```
77 buffer              display-2 pic x(256).
77 file-name           display-2 pic x(256).
call "xsm_fi_path" using file-name giving buffer.
```

DESCRIPTION

Use this function to find the full path name of a file. The file may be a screen or any other type of file. The file's full path name is returned in `buffer`.

The file name is first sought in the current directory. If that fails, the path given to `xsm_initcrt` is checked. Finally the path defined by `SMPATH` is searched.

RETURNS

0 if the file cannot be found in any path.
Else, The path is returned in `buffer`.

finquire

obtain information about a field

SYNOPSIS

```
copy "smglobs.incl.cobol".

77 field-number      pic S(9)9 comp-5.
77 which             pic S(9)9 comp-5.
77 value             pic S(9)9 comp-5.
call "xsm_finquire" using field-number, which giving value.
```

DESCRIPTION

Use this function to obtain various information about a field. The variable `which` is a value that specifies the particular piece of information desired.

Values for `which` are defined in the file `smglobs.incl.cobol`. The following values are available:

Value	*Meaning*
FD-LINE	Line that field is on.
FD-COLM	Column of field's first position.
FD-ATTR	Field attributes (see `smattrib.incl.cobol`).
FD-LENG	Onscreen field length.
FD-ASIZE	Onscreen array size (1 if scalar).
FD-ELT	Onscreen element number.
FD-SHLENG	Shiftable length.
FD-SHINCR	Shift increment.
FD-SHOFS	Current shift offset (number of positions field has been shifted; 0 if shifted to left edge).
FD-SCINCR	Scrolling increment (for Next/Prev page keys).
FD-SCFLAG	Scrolling array circular? (T/F).

Value	*Meaning*
FD-SCATTR	Scrolling occurrence display attributes set with xsm_i_achg; zero if onscreen element attributes is to be used. For xsm_i_finquire variant only.
FD-FELT	First onscreen occurrence of scrolling array (1 if scrolled to top).

RETURNS

The value of which if found.
0 otherwise.

VARIANTS

```
call "xsm_e_finquire" using field-name, element, which giving
     value.
call "xsm_i_finquire" using field-name, occurrence, which
     giving value.
call "xsm_n_finquire" using field-name, which giving value.
call "xsm_o_finquire" using field-number, occurrence, which
     giving value.
```

RELATED FUNCTIONS

```
call "xsm_gp_inquire" using group-name, which giving value.
call "xsm_inquire" using which giving value.
call "xsm_iset" using which, newval giving value.
call "xsm_pinquire" using which giving buffer.
call "xsm_pset" using which, newval giving buffer.
```

fldno

get the field number of an array element or occurrence

SYNOPSIS

```
77 field-name          display-2 pic x(256).
77 field-number        pic S(9)9 comp-5.
call "xsm_n_fldno" using field-name giving field-number.
```

DESCRIPTION

NOTE: This function only exists in the `e_`, `i_`, `n_`, and `o_` variations. There is NO `xsm_fldno` since this function determines the field number given other information.

The `e_` variant returns the field number of an array element specified by `field-name` and `element`. If `element` is zero, then `xsm_e_fldno` returns the field number of the named field, or the base element of the named array.

The `i_` and `o_` variants return the number of the field containing the specified occurrence if the occurrence is onscreen, or 0 if the occurrence is offscreen.

The `n_` variant returns the field number of a field specified by name, or the base field number of an array specified by name.

RETURNS

0 if the name is not found, if the element number exceeds 1 and the named field is not an array, or if the occurrence is offscreen.

Otherwise, returns an integer between 1 and the maximum number of fields on the current screen that represents the field number.

VARIANTS

```
call "xsm_e_fldno" using field-name, element giving
       field-number.
call "xsm_i_fldno" using field-name, occurrence giving
       field-number.
call "xsm_o_fldno" using field-number, occurrence giving
       field-number.
```

flush

flush delayed writes to the display

SYNOPSIS

```
call "xsm_flush".
```

DESCRIPTION

This function performs delayed writes and flushes all buffered output to the display. It is called automatically via `xsm_input` whenever the keyboard is opened and there are no keystrokes available, *i.e.* typed ahead.

Calling this routine indiscriminately can significantly slow execution. As it is called whenever the keyboard is opened, the display is always guaranteed to be in sync before data entry occurs; however, if you want timed output or other non–interactive display, use of this routine will be necessary.

RELATED FUNCTIONS

```
call "xsm_flush".
call "xsm_rescreen".
```

form

display a screen as a form

SYNOPSIS

```
77 screen-name          display-2 pic x(256).
77 status               pic S(9)9 comp-5.
call "xsm_r_form" using screen-name giving status.


copy "myscreen.incl.cobol".

77 status               pic S(9)9 comp-5.
call "xsm_d_form" using SCREEN-ADDRESS giving status.


77 lib-desc             pic S(9)9 comp-5.
77 screen-name          display-2 pic x(256).
77 status               pic S(9)9 comp-5.
call "xsm_l_form" using lib-desc, screen-name giving status.
```

DESCRIPTION

This set of functions is primarily intended to be used by developers who are writing their own executive. These functions do not update the form stack, so it is generally not a good idea to use them with the **JAM** Executive. To open a form while under the control of the **JAM** Executive, use a **JAM** control string or `xsm_jform`.

These functions display the named screen as a base form. Bringing up a screen as a form with `xsm_d_form`, `xsm_l_form`, `xsm_r_form` causes the previously displayed form and windows to be discarded, and their memory freed. The new screen is displayed with its upper left-hand corner at the extreme upper left of the display (position (0, 0)).

If an error occurs a return of –1 or –2 means that the previously displayed form is still displayed and may be used. Other negative return codes indicate that the display is undefined. The caller should display another form before using Screen Manager functions.

When you use `xsm_r_form` the named screen is sought first in the memory-resident screen list, and if found there is displayed using `xsm_d_form`. It is next sought in all the open screen libraries, and if found is displayed using `xsm_l_form`. Next it is sought on disk in the current directory; then under the path supplied to `xsm_initcrt`; then in all the paths in the setup variable `SMPATH`. If any path exceeds 80 characters, it is skipped. If the entire search fails, this function displays an error message and returns.

You may save processing time by using `xsm_d_form` to display screens that are memory–resident. Use `bin2cob` to convert screens from disk files, which you can modify using `jxform`, to program data structures you can compile into your application. A memory–resident screen is never altered at run–time, and may therefore be made shareable on systems that provide for sharing read–only data. `xsm_r_form` can also display memory–resident screens, if they are properly installed using `xsm_formlist`. Memory–resident screens are particularly useful in applications that have a limited number of screens, or in environments that have a slow disk (*e.g.* MS–DOS). `SCREEN-ADDRESS` is the address of the screen in memory.

You may also save processing time by using `xsm_l_form` to display screens that are in a library. A library is a single file containing many screens (and/or JPL modules and keysets). You can assemble one from individual screen files using the utility `formlib`. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on the number of paths searched.

The library descriptor, `lib-desc`, is an integer returned by `xsm_l_open`, which you must call before trying to read any screens from a library. Note that `xsm_r_form` also searches any open libraries.

To display a window use `xsm_r_at_cur`, `xsm_r_window`, or one of their variants.

RETURNS

0 if no error occurred
–1 if the screen file's format is incorrect; previous form still displayed and available
–2 if the screen cannot be found or the maximum allowable number of files is already open; previous form still displayed and available
–4 if, after the screen has been cleared, the screen cannot be successfully displayed because of a read error;
–5 if, after the screen was cleared, the system ran out of memory;

RELATED FUNCTIONS

```
call "xsm_r_window" using screen-name, start-line, start-column
       giving status.
call "xsm_r_at_cur" using screen-name giving status.
```

formlist

update list of memory–resident files

SYNOPSIS

```
copy "myform.incl.cobol".

77 name                display-2 pic x(256).
77 status              pic S(9)9 comp-5.
call "xsm_formlist" using name, address giving status.
```

DESCRIPTION

This function adds a JPL module, keyset, or screen to the memory resident form list. Each member of the list is a structure giving the name of the JPL module, screen, or keyset, as a character string, and its address in memory. This function is commonly called from `main`. It can be called any number of times from an application program to augment to the memory resident list.

To make a JPL module, keyset, or screen memory resident, you can use the `bin2cob` utility to create a COBOL record initialized with the binary content of the object. You must then copy the record into the application executable.

RETURNS

–1 if insufficient memory is available for the new list;
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_rmformlist.
```

ftog

convert field references to group references

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 group-occurrence   pic S(9)9 comp-5.
77 buffer             display-2 pic x(256)
call "xsm_ftog" using field-number, group-occurrence giving
       buffer.
```

DESCRIPTION

This function converts field references to group references. Use `xsm_i_gtof` to convert them back.

This function returns the name of the group containing the referenced field and inserts its group occurrence number into the address of `occurrence`.

RETURNS

The group name if found and indirectly through `group-occurrence` the group occurrence number.
0 otherwise and `group-occurrence` is unchanged.

VARIANTS

```
call "xsm_e_ftog" using field-name, element, group-occurrence
       giving buffer.
call "xsm_i_ftog" using field-name, occurrence,
       group-occurrence giving buffer.
call "xsm_n_ftog" using field-name, group-occurrence giving
       buffer.
call "xsm_o_ftog" using field-number, occurrence,
       group-occurrence giving buffer.
```

RELATED FUNCTIONS

```
call "xsm_i_gtof" using group-name, group-occurrence,
       occurrence giving field-number.
```

ftype

get the data type and precision of a field

SYNOPSIS

```
copy "smedits.incl.cobol".

77 field-number       pic S(9)9 comp-5.
77 precision-ptr      pic S(9)9 comp-5.
77 type               pic S(9)9 comp-5.
call "xsm_ftype" using field-number, precision-ptr giving type.
```

DESCRIPTION

This function analyzes the edits of a field or LDB entry, and returns data type information. First the "type" (FTYPE) edit is checked, then the "currency" edit, the "date/time" edit, and finally the "character" edit.

Note that this differs from the functionality of `xsm_rdstruct`, `xsm_wrtstuct`, `xsm_rrecord`, and `xsm_wrecord`. These functions only test the type and character edits. They use the currency edit only to determine the precision of a numeric field that has no type edit.

This function returns an integer containing the data type code, plus any applicable flags. The data type codes and flags are detailed in the tables below.

Data Type Code	*Meaning*
FT-CHAR	Type edit is *char string*; or character edit is *unfiltered, letters only, alphanumeric*, or *regular expression*
FT-INT	Type edit is *int*
FT-UNSIGNED	Type edit is *unsigned int*; or character edit is *digit*
FT-SHORT	Type edit is *short int*
FT-LONG	Type edit is *long int*
FT-FLOAT	Type edit is *float*
FT-DOUBLE	Type edit is *double*; or character edit is *numeric*
FT-ZONED	Type edit is *zoned dec.*

Data Type Code	*Meaning*
FT-PACKED	Type edit is *packed dec.*
DT-YESNO	Character edit is *yes/no*
DT-CURRENCY	Currency edit
DT-DATETIME	Date/time edit

Flag	*Meaning*
DF-NULL	Null edit
DF-REQUIRED	Data required edit (not applicable to LDB)
DF-WRAP	Word wrap edit
DF-OMIT	Type edit is *omit.*

To determine the data type code, check this integer for each flag in the fashion of the example field function shown on page 14, starting with DF-OMIT and working up the list.. The value remaining will be the data type code.

Note that FT-OMIT is not listed as one of the data types. A field that has the type edit *omit* will return the data type determined by any of the other edits, as well as a flag indicating that it has the *omit* type edit.

The function will put the precision of float, double and currency values in the precison-ptr argument.

RETURNS

major data type code plus any applicable flags (see tables above).
0 if field is not found

VARIANTS

```
call "xsm_n_ftype" using field-number, precision-ptr giving
      type.
```

fval

force field validation

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_fval" using field-number giving status.
```

DESCRIPTION

This function performs all validations on the indicated field or occurrence, and returns the result. If the field is protected against validation, the checks are not performed and the function returns 0; see `xsm_aprotect`. Validations are done in the order listed below. Some will be skipped if the field is empty, or if its VALIDED bit is already set (implying that it has already passed validation).

Validation	*Skip if valid*	*Skip if empty*
required	y	n
must fill	y	y
regular expression	y	y
range	y	y
check–digit	y	y
date or time	y	y
table lookup	y	y
currency format	y	n*
math expresssion	n	n
field validation	n	n
JPL function	n	n

* The currency format edit contains a skip–if–empty flag; see the Author's Guide.

If you need to force a skip–if–empty validation, make the field required. A field with embedded punctuation must contain at least one non–blank non–punctuation character in or-

der to be considered non–empty; otherwise any non blank character makes the field non–empty.

Math expressions, JPL functions and field validation functions are never skipped, since they can alter fields other than the one being validated.

Field validation is performed automatically within `xsm_input` when the cursor exits a field via the `TAB` or `NL` logical keys. All fields on a screen are validated when `XMIT` is pressed (see `xsm_s_val`). Application programs need call this function only to force validation of other fields.

RETURNS

–2 if the field or occurrence specification is invalid;
–1 if the field fails any validation;
0 otherwise.

VARIANTS

```
call "xsm_e_fval" using array-name, element giving status.
call "xsm_i_fval" using field-name, occurrence giving status.
call "xsm_n_fval" using field-name giving status.
call "xsm_o_fval" using field-number, occurrence giving status.
```

RELATED FUNCTIONS

```
call "xsm_n_gval" using group-name giving status.
call "xsm_s_val" giving status.
```

getcurno

get current field number

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
call "xsm_getcurno" giving field-number.
```

DESCRIPTION

This function returns the number of the field in which the cursor is currently positioned. The field number ranges from 1 to the total number of fields in the screen.

RETURNS

Number of the current field, or
0 if the cursor is not within a field.

RELATED FUNCTIONS

```
call "xsm_occur_no" giving occurrence.
```

getfield

copy the contents of a field

SYNOPSIS

```
77 buffer              display-2 pic x(256).
77 field-number        pic S(9)9 comp-5.
77 length              pic S(9)9 comp-5.
call "xsm_getfield" using buffer, field-number giving length.
```

DESCRIPTION

This function copies the data found in `field-number` to `buffer`. Leading blanks in right-justified fields and trailing blanks in left-justified fields are not copied. The variants that reference a field by name will attempt to get data from the corresponding LDB entry if there is no such field on the screen (except that the order is reversed during screen entry/exit processing).

Responsibility for providing a buffer large enough for the field's contents rests with the calling program. This should be at least one greater than the maximum length of the field, taking shifting into account.

In variants that take `name` as an argument, either the name of a field or a group may be used. In the case of groups, `xsm_isselected` is preferred to `xsm_getfield` for determining whether or not a group occurrence is selected. If `xsm_n_getfield` is called on a radio button, the value in `buffer` will be the occurrence number of the selected item. If `xsm_i_getfield` is called on a checklist, the value in the first occurrence of the array will be the number of the first selected item in the group, the value in the second occurrence will be the number of the next selected item in the group and so on. If a checklist has, for example, three items selected, the fourth array occurrence will be empty.

Note that the order of arguments to this function is different from that to the related function `xsm_putfield`.

RETURNS

The total length of the field's contents, or
-1 if the field cannot be found.

VARIANTS

```
call "xsm_e_getfield" using buffer, name, element giving
       length.
call "xsm_i_getfield" using buffer, name, occurrence giving
       length.
```

```
call "xsm_n_getfield" using buffer, name giving length.
call "xsm_o_getfield" using buffer, field-number, occurrence
      giving length.
```

RELATED FUNCTIONS

```
call "xsm_isselected" using group-name, group-occurrence giving
      status.
call "xsm_putfield" using field-number, data giving status.
```

getjctrl

get control string associated with a key

SYNOPSIS

```
copy "smkeys.incl.cobol".

77 key                pic S(9)9 comp-5.
77 default            pic S(9)9 comp-5.
77 buffer             display-2 pic x(256)
call "xsm_getjctrl" using key, default giving buffer.
```

DESCRIPTION

Each **JAM** screen contains a table of control strings associated with function keys. **JAM** also maintains a default table of keys and control strings, which take effect when the current screen has no control string for a function key you press. This table enables you to define system–wide actions for keys. It is initialized from `SMINICTRL` setup variables. See the section on setup in the Configuration Guide for further information.

This function searches one of the tables for `key`, a logical key value found in `smkeys.incl.cobol`, and returns a the associated control string. If `default` is zero, the table for the current screen is searched; otherwise, the system–wide table is searched.

RETURNS

The control string
0 if none is found.

RELATED FUNCTIONS

```
call "xsm_putjctrl" using key, control-string, default giving
      status.
```

getkey

get logical value of the key hit

SYNOPSIS

```
copy "smkeys.incl.cobol".

77 key                pic S(9)9 comp-5.
call "xsm_getkey" giving key.
```

DESCRIPTION

This function gets and interprets keyboard input and returns the logical value to the calling program. Normal characters are returned unchanged. Logical keys are interpreted according to a key translation file for the particular terminal you are using. See the Keyboard Input section in this guide, the Key Translation section in the Configuration Guide, and the `modkey` section in the Utilities Guide. `xsm_getkey` is normally not needed for application programming, since it is called by `xsm_input`.

Logical keys include TRANSMIT, EXIT, HELP, LOCAL PRINT, arrows, data modification keys like INSERT and DELETE CHAR, user function keys PF1 through PF24, shifted function keys SPF1 through SPF24, and others. Defined values for all are in `smkeys.incl.cobol`. A few logical keys, such as LOCAL PRINT and RESCREEN, are processed locally in `xsm_getkey` and not returned to the caller.

There is another function called `xsm_ungetkey`, which pushes logical key values back on the input stream for retrieval by `xsm_getkey`. Since all **JAM** input routines call `xsm_getkey`, you can use it to generate any input sequence automatically. When you use it, calls to `xsm_getkey` will not cause the display to be flushed, as they do when keys are read from the keyboard.

There are a number of user-installed functions that may be called by `xsm_getkey`. For further information see the section on installing functions in the Programmer's Guide.

Finally, there is a mechanism for detecting an externally established abort condition, essentially a flag, which causes **JAM** input functions to return to their callers immediately. The present function checks for that condition on each iteration, and returns the ABORT key if it is true. See `xsm_isabort`.

Application programmers should be aware that **JAM** control strings are not executed within this function, but at a higher level within the **JAM** run-time system (i.e., functions that call `xsm_getkey`. If you call this function, do not expect function key control strings to work.

The multiplicity of calls to user functions in `xsm_getkey` makes it a little difficult to see how they interact, which take precedence, and so forth. In an effort to clarify the process, we present an outline of `xsm_getkey`. The process of key translation is deliberately omitted, for the sake of clarity; that algorithm is presented separately, in the keyboard translation section of the Programmer's Guide.

***Step 1

- If an abort condition exists, return the ABORT key.
- If there is a key pushed back by ungetkey, return that.
- If playback is active and a key is available, take it directly to Step 2; otherwise read and translate input from the keyboard. When the keyboard is read, then the asynchronous function (if one is installed) is called during periods of keyboard inactivity.

*** Step 2

- Pass the key to the keychange function. If that function says to discard the key, go back to Step 1; otherwise if an abort condition exists, return the ABORT key.
- If recording is active, pass the key to the recording function.

*** Step 3

- If the routing table says the key is to be processed locally, do so.
- If the routing table says to return the key, return it; otherwise, go back to Step 1.
- If the key is a soft key, return its logical value.

RETURNS

The standard ASCII value of a displayable key; a value greater than 255 (FF hex) for a key sequence in the key translation file.

RELATED FUNCTIONS

```
call "xsm_keyfilter" using flag giving old-flag.
call "xsm_ungetkey" using key giving return-value.
```

gofield

move the cursor into a field

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_gofield" using field-number giving status.
```

DESCRIPTION

Positions the cursor to the first enterable position of `field-number`. If the field is shiftable, it is reset.

In a right–justified field, the cursor is placed in the rightmost position and in a left–justified field, in the leftmost. In either case, if the field has embedded punctuation, the cursor goes to the nearest position not occupied by a punctuation character. Use `xsm_off_gofield` to place the cursor in position other than that of the first character of a field.

When called to position the cursor in a scrollable array, `xsm_o_gofield` and `xsm_i_gofield` return an error if the occurrence number passed exceeds by more than 1 the number of allocated occurrences in the specified array. If the desired occurrence is offscreen, it is scrolled on–screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
call "xsm_e_gofield" using field-name, element giving status.
call "xsm_i_gofield" using field-name, occurrence giving
       status.
call "xsm_n_gofield" using field-name giving status.
call "xsm_o_gofield" using field-number, occurrence giving
       status.
```

RELATED FUNCTIONS

```
call "xsm_off_gofield" using field-number, offset giving
       status.
```

gp_inquire

obtain information about a group

SYNOPSIS

```
copy "smglobs.incl.cobol".

77 group-name          display-2 pic x(256).
77 which               pic S(9)9 comp-5.
77 value               pic S(9)9 comp-5.
call "xsm_gp_inquire" using group-name, which giving value.
```

DESCRIPTION

Use this function to obtain various information about group. The variable `which` is a value that specifies the particular piece of information desired.

Values for `which` are defined in the file `smglobs.incl.cobol`. They are:

Value	*Meaning*
GP-NOCCS	Number of occurrences in the group (sum of number of occurrences of all fields/arrays in group)
GP-FLAGS	Flags

RETURNS

The value of `which`, if found, or
-1 otherwise.

gtof

convert a group name and index into a field number and occurrence

SYNOPSIS

```
77 group-name          display-2 pic x(256).
77 group-occurrence    pic S(9)9 comp-5.
77 occurrence          pic S(9)9 comp-5.
77 field-number        pic S(9)9 comp-5.
call "xsm_i_gtof" using group-name, group-occurrence,
       occurrence giving field-number.
```

DESCRIPTION

NOTE: This function only exists in the `i_` variation. There is no `xsm_gtof` since groups cannot be referenced by number.

Use this function to convert a group name and group-occurrence into a field number and occurrence. The variable `group-name` is the name of the group and `group-occurrence` is the specific field within the group.

The function returns the field number of the referenced field and inserts the occurrence number into the memory location addressed by `occurrence`.

Using this function allows you to use other JAM library routines to manipulate group fields by converting group references into field references. For instance, if you wanted to access text from a specific field within a group you would need to use `xsm_i_gtof` to get the field and occurrence number before you could use the function `xsm_o_getfield` to retrieve the text.

RETURNS

The field number if found.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_ftog" using field-number, group-occurrence giving
       buffer.
```

gval
force group validation

SYNOPSIS

```
77 group-name          display-2 pic x(256).
77 status              pic S(9)9 comp-5.
call "xsm_n_gval" using group-name giving status.
```

DESCRIPTION

NOTE: This function only exists in the `xsm_n_gval` variation. There is no `xsm_gval` since groups cannot be referenced by number.

Use this function to force the execution of a group's validation function. Use `xsm_s_val` to validate all fields and groups on the screen.

RETURNS

-1 if the group fails any validation.
-2 if the group name is invalid.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_fval" using field-number giving status.
call "xsm_s_val" giving status.
```

gwrap

get the contents of a wordwrap array

SYNOPSIS

```
77 buffer              display-2 pic x(256).
77 field-number        pic S(9)9 comp-5.
77 buffer-length       pic S(9)9 comp-5.
77 length              pic S(9)9 comp-5.
call "xsm_gwrap" using buffer, field-number, buffer-length
       giving length.
```

DESCRIPTION

This function copies the contents of the array specified by `field-number`, one occurrence at a time, into `buffer`, up to the size specified by `buffer-length`. A space is inserted before every non–empty occurrence, except the first.

The variant `xsm_o_gwrap` copies the contents of the array, beginning with the specified occurrence.

RETURNS

The length of transferrable data. If this is greater than `buffer-length`, then the data was truncated.

–1 if the field number is invalid or `buffer-length` is ≤ 0.

VARIANTS

```
call "xsm_o_gwrap" using buffer, field-number, occurrence,
       buffer-length giving status.
```

RELATED FUNCTIONS

```
call "xsm_pwrap" using field-number, text giving status.
```

hlp_by_name
display help window

SYNOPSIS

```
77 help-screen        display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_hlp_by_name" using help-screen giving status.
```

DESCRIPTION

The named screen is displayed and processed as a normal help screen, including input processing for the current field (if any).

Refer to the Author's Guide for instructions on how to create various kinds of help screens and for details of the behaviour of help screens.

RETURNS

-1 if screen is not found or other error;
1 if data copied from help screen to underlying field;
0 otherwise.

home

home the cursor

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
call "xsm_home" giving field-number.
```

DESCRIPTION

This function moves the cursor to the first enterable position of the first tab–unprotected field on the screen. If the screen has no tab–unprotected fields, the cursor is moved to the first line and column of the topmost screen. However, if you are using the JAM Executive, the cursor may not be visible if there are no tab–unprotected fields.

The cursor will be put into a tab–protected field if it occupies the first line and column of the screen and there are no tab–unprotected fields.

This function doesn't immediately trigger field entry, exit, or validation processing. Processing is based on the cursor position when control returns to `xsm_input`.

When the JAM logical key HOME is hit, `xsm_home` is called.

RETURNS

The number of the field in which the cursor is left, or
0 if the form has no unprotected fields and the home position is not in a protected field.

RELATED FUNCTIONS

```
call "xsm_backtab".
call "xsm_gofield" using field-number giving status.
call "xsm_last".
call "xsm_nl".
call "xsm_tab".
```

i_

variants that take a field name and occurrence number

SYNOPSIS

```
77 field-name          display-2 pic x(256).
77 occurrence          pic S(9)9 comp-5.
call "xsm_i_..." using field-name, occurrence, ....
```

DESCRIPTION

The `i_` variants each refer to data by field name and occurrence number. An occurrence is a slot within an array in which data may be stored. Occurrences may be either on or off–screen. Since JAM treats an individual field as an array with one field, even a single non–scrolling field is considered to have one occurrence. The JAM library contains routines that allow you to manipulate individual occurrences during run–time.

If `occurrence` is zero, the reference is always to the current contents of the named field, or of the base field of the named array.

For the description of a particular function, look under the related function without `i_` in its name. For example, `xsm_i_amt_format` is described under `xsm_amt_format`.

If the named field is not part of the screen currently being displayed, these functions will attempt to retrieve or change its value in the local data block.

ininames

record names of initial data files for local data block

SYNOPSIS

```
77 name-list          display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_ininames" using name-list giving status.
```

DESCRIPTION

Use this routine to set up a list of initialization files for local data block entries. The file names in the single string `name-list` should be separated by commas, semicolons or blanks. There may be up to ten file names. You may achieve the same effect by defining the `SMININAMES` variable in your setup file to the list of names. See setup files in the Configuration Guide and the Data Dictionary chapter of the Author's Guide for details.

The files contain pairs of names and values, which are used to initialize local data block entries by `xsm_ldb_init`. This function is called during JAM initialization, so `xsm_ininames` should be called before then. White space in the initialization files is ignored, but we suggest a format like the following:

```
"emperor"      "Julius Caesar"
"lieutenant"   "Mark Antony"
"assassin[1]"  "Brutus"
"assassin[2]"  "Cassius"
```

Entries of all scopes may be freely mixed within all files. We recommend, however, that entries be grouped in files by scope if you are planning to use `xsm_lreset`. Use `xsm_lreset` to clear all entries of a given scope before reinitializing them from a single file.

RETURNS

-5 if insufficient memory is available to store the names;
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_ldb_init".
call "xsm_lreset" using file-name, scope giving status.
```

initcrt

initialize the display and **JAM** data structures

SYNOPSIS

```
77 path                display-2 pic x(256).
call "xsm_initcrt" using path.

77 path                display-2 pic x(256).
call "xsm_jinitcrt" using path.

77 path                display-2 pic x(256).
call "xsm_jxinitcrt" using path.
```

DESCRIPTION

The function `xsm_initcrt` is intended for use only with a user–written executive. It is called automatically by the **JAM** Executive.

`xsm_initcrt` must be called at the beginning of screen handling, that is, before any screens are displayed or the keyboard opened for input to a **JAM** screen. Functions that set options, such as `xsm_option`, and those that install functions or configuration files such as `xsm_uinstall` or `xsm_vinit`, are the only kind that may be called before `xsm_initcrt`.

The argument `path` is a directory to be searched for screen files by `xsm_r_window` and variants. First the file is sought in the current directory; if it is not there, it is sought in the path supplied to this function. If it is not there either, the paths specified in the environment variable `SMPATH` (if any) are tried. The `path` argument *must* be supplied. If all forms are in the current directory, or if (as JYACC suggests) all the relevant paths are specified in `SMPATH`, an empty string may be passed. After setting up the search path, `xsm_initcrt` performs several initializations:

1. It calls a user–defined initialization routine.
2. It determines the terminal type, if possible by examining the environment (`TERM` or `SMTERM`), otherwise by asking the user.
3. It executes the setup files defined by the environment variables `SMVARS` and `SMSETUP`, and reads in the binary configuration files (message, key, and video) specific to the terminal.
4. It allocates memory for a number of data structures shared among **JAM** library functions.

5. If supported by the operating system, keyboard interrupts are trapped to a routine that clears the display and exits.
6. It initializes the operating system display and keyboard channels, and clears the display.

The functions `xsm_jinitcrt` and `xsm_jxinitcrt` are called by `jmain.cobol` and `jxmain.cobol` respectively for applications that use the JAM Executive. They, in turn, call `xsm_initcrt`.

RELATED FUNCTIONS

```
call "xsm_resetcrt".
call "xsm_jresetcrt".
call "xsm_jxresetcrt".
```

input

open the keyboard for data entry and menu selection

SYNOPSIS

```
copy "smumisc.incl.cobol".

77 initial-mode       pic S(9)9 comp-5.
77 key                pic S(9)9 comp-5.
call "xsm_input" using initial-mode giving key.
```

DESCRIPTION

This routine is only used if you are writing your own executive. Use `xsm_input` to open the keyboard for either data entry or menu selection.

You specify which mode you wish to be in with the argument `initial-mode`. Possible choices are defined in `smumisc.incl.cobol`. They are:

- IN-AUTO JAM checks whether you specified the screen to begin menu mode or data entry mode (See Author's Guide).
- IN-DATA Start in data entry mode.
- IN-MENU Start in menu mode.

In most cases you will want to use IN-AUTO mode. Use IN-DATA or IN-MENU if you wish to override the setting that you specified via the Screen Editor.

This routine calls `xsm_getkey` to get and interpret keyboard entry. While in data entry mode ASCII data is entered into fields on the screen, subject to any restrictions or edits that were defined for the fields. The routine returns to the calling program when it encounters a logical key, when a "return entry" field is filled or tabbed from, or a key with the return bit set in the routing table.

If the logical value returned by `xsm_getkey` is TRANSMIT, EXIT, HELP, or a cursor position key, the processing is determined by a routing table. The routing options are set with `xsm_keyoption`. See `xsm_keyoption` for more information.

This function replaces version 4.0 `xsm_choice`, `xsm_menu_proc`, and `xsm_openkeybd`. These functions only exist in your version 5.0 library for backward compatibility. We strongly suggest that you do not use them in the future.

RETURNS

The key hit by the end-user that terminated the call to `xsm_input`, or the first character of the selected menu item.

inquire

obtain value of a global integer variable

SYNOPSIS

```
copy "smglobs.incl.cobol".

77 which              pic S(9)9 comp-5.
77 value              pic S(9)9 comp-5.
call "xsm_inquire" using which giving value.
```

DESCRIPTION

This function is used to obtain the current integer value of a global variable. The desired variable is specified by `which`. If the value of `which` is a true/false (the flag is on or off) value then `xsm_inquire` returns `1` for true and `0` for false. If you wish to modify a global integer value use `xsm_iset`. The permissible values for `which` are defined in `smglobs.incl.cobol`. The following values are available:

Value	*Meaning*
`I-NODISP`	In non-display mode? (T/F). Initially FALSE, setting TRUE causes no further changes to the actual display, although **JAM**'s internal screen image is kept up to date. This was release 4's `sm-do-not-display` flag.
`I-INSMODE`	In insert mode? (T/F).
`I-INXFORM`	In **JAM** screen editor? (T/F) Field validation routines are generally still called when in editor; they can check this flag to disable certain features.
`I-MXLINES`	Number of lines available for use by **JAM** on the hardware display.
`I-MXCOLMS`	Number of columns available for use by **JAM** on the hardware display.
`I-NLINES`	Maximum number of lines available on the current screen, not including the status line.
`I-NCOLMS`	Maximum number of columns available on the current screen, not including the status line.
`I-INHELP`	Help screen is currently displayed? (T/F)

Value	*Meaning*
I-BSNESS	Screen manager is in control of display? (T/F). (Replaces rel. 4 inbusiness function).
I-BLKFLGS	Block mode is turned on? (T/F)
SC-VFLINE	First screen line of viewport (0–based).
SC-VFCOLM	First screen column of viewport (0–based).
SC-VNLINE	Number of lines in viewport.
SC-VNCOLM	Number of columns in viewport.
SC-VOLINE	Line offset of viewport.
SC-VOCOLM	Column offset of viewport.
SC-NLINE	Number of lines in screen.
SC-NCOLM	Number of columns in screen.
SC-CLINE	Current line number in screen.
SC-CCOLM	Current column number in screen.
SC-NFLDS	Number of fields on screen.
SC-NGRPS	Number of groups on screen.
SC-BKATTR	Background attributes of screen.
SC-BDCHAR	Border character of screen.
SC-BDATTR	Border attributes of screen.

RETURNS

If the argument corresponds to an integer global variable, the current value of that variable is returned.

1 true, flag is set to on.
0 false, flag is set to off.
–1 otherwise.

RELATED FUNCTIONS

```
call "xsm_finquire" using field-number, which giving value.
call "xsm_gp_inquire" using group-name, which giving value.
call "xsm_iset" using which, newval giving value.
call "xsm_pinquire" using which giving buffer.
call "xsm_pset" using which, newval giving buffer.
```

intval

get the integer value of a field

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 value              pic S(9)9 comp-5.
call "xsm_intval" using field-number giving value.
```

DESCRIPTION

This function returns the integer value of the data contained in the field specified by `field-number`. Any punctuation characters in the field, except a leading plus or minus sign, are ignored.

RETURNS

The integer value of the specified field.
0 if the field is not found.

VARIANTS

```
call "xsm_e_intval" using field-name, element giving value.
call "xsm_i_intval" using field-name, occurrence giving value.
call "xsm_n_intval" using field-name giving value.
call "xsm_o_intval" using field-number, occurrence giving
      value.
```

RELATED FUNCTIONS

```
call "xsm_itofield" using field-number, value giving status.
```

ioccur

insert blank occurrences into an array

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 occurrence         pic S(9)9 comp-5.
77 count              pic S(9)9 comp-5.
77 lines-inserted     pic S(9)9 comp-5.
call "xsm_o_ioccur" using field-number, occurrence, count
       giving lines-inserted.
```

DESCRIPTION

NOTE: This function only exists in the `i_` and `o_` variations. There is no `xsm_ioccur`, since this function applies only to arrays.

Inserts `count` blank occurrences before the specified occurrence, moving that occurrence and all following occurrences down. If inserting that many would move an occurrence past the end of its array, fewer will be inserted. If the array is scrollable, then this function may allocate up to `count` new occurrences. This function never increases the maximum number of occurrences an array can contain; `xsm_sc_max` does that. If `count` is negative, occurrences will be deleted instead, subject to limitations described in the page for `xsm_doccur`. In addition, this function never inserts more blank occurrences than the number of blank occurrences following the last non–blank occurrence (that is, it won't push data off the end of an array).

If `occurrence` is zero, the occurrence used is that of `field-number`. If `occurrence` is nonzero, however, it is taken relative to the first field of the array in which `field-number` occurs.

Any clearing–unprotected synchronized arrays will have the same operations performed on them as the referenced array. Synchronized arrays that are protected from clearing will remain constant. Therefore, a protected array may be used to number a list of data stored in a non–protected synchronized array as it grows and shrinks.

This function is normally bound to the INSERT LINE key.

RETURNS

–1 if the field or occurrence number is out of range.
–3 if insufficient memory is available.
otherwise, the number of occurrences actually inserted (zero or more).

VARIANTS

```
call "xsm_i_ioccur" using field-name, occurrence, count giving
      lines-inserted.
```

is_no

test field for no

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_is_no" using field-number giving status.
```

DESCRIPTION

The first character of the field contents specified by `field-number` is compared with the first letter of the SM-NO entry in the message file, ignoring case. If they match this function will return a 1 for true. If they do not match for any reason, the function returns a 0 for failure. There is no way to tell if the failure is due to a Y in the field or because of some other problem. If you wish to check for a Y response use `xsm_is_yes`.

This function is ordinarily used with one-letter fields possessing the yes/no character edit. In this case, the only characters allowed in the field are y, n, or space (which means n). Unlike other functions, `xsm_is_no` does *not ignore leading blanks*.

RETURNS

1 if the field's first character matches the first character of the SM-NO entry in the message file.
0 otherwise.

VARIANTS

```
call "xsm_e_is_no" using field-name, element giving status.
call "xsm_i_is_no" using field-name, occurrence giving status.
call "xsm_n_is_no" using field-name giving status.
call "xsm_o_is_no" using field-number, occurrence giving
      status.
```

RELATED FUNCTIONS

```
call "xsm_is_yes" using field-number giving status.
```

is_yes
test field for yes

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_is_yes" using field-number giving status.
```

DESCRIPTION

The first character of the field contents specified by `field-number` is compared with the first letter of the `SM-YES` entry in the message file, ignoring case. If they match this function will return a 1 for true. If they do not match for any reason, the function returns a 0 for failure. There is no way to tell if the failure is due to an `N` in the field or because of some other problem. If you wish to check for an `N` response use `xsm_is_no`.

This function is ordinarily used with one-letter fields possessing the yes/no character edit. In this case, the only characters allowed in the field are `y`, `n`, or space (which means `n`). Unlike other functions, `xsm_is_yes` does *not ignore leading blanks*.

RETURNS

1 if the field's first character matches the first character of the `SM-YES` entry in the message file.
0 otherwise.

VARIANTS

```
call "xsm_e_is_yes" using field-name, element giving status.
call "xsm_i_is_yes" using field-name, occurrence giving status.
call "xsm_n_is_yes" using field-name giving status.
call "xsm_o_is_yes" using field-number, occurrence giving
      status.
```

RELATED FUNCTIONS

```
call "xsm_is_no" using field-number giving status.
```

isabort

test and set the abort control flag

SYNOPSIS

```
copy "smumisc.incl.cobol".

77 flag                pic S(9)9 comp-5.
77 old-flag            pic S(9)9 comp-5.
call "xsm_isabort" using flag giving OLD-FLAG.
```

DESCRIPTION

Use `xsm_isabort` to set the abort flag to the value of `flag`, and return the old value. `flag` must be one of the following as defined in `smumisc.incl.cobol`:

Flag	*Meaning*
ABT-ON	set abort flag
ABT-OFF	clear abort flag
ABT-DISABLE	turn abort reporting off
ABT-NOCHANGE	do not alter the flag

Abort reporting is intended to provide a quick way out of processing in the JAM library, which may involve nested calls to `xsm_input`. The triggering event is the detection of an abort condition by `xsm_getkey`, either an ABORT keystroke or a call to this function with `ABT-ON` (such as from an asynchronous function).

This function enables application code to verify the existence of an abort condition by testing the flag, as well as to establish one.

RETURNS

The previous value of the abort flag.

iset

change value of integer global variable

SYNOPSIS

```
copy "smglobs.incl.cobol".

77 which              pic S(9)9 comp-5.
77 newval             pic S(9)9 comp-5.
77 value              pic S(9)9 comp-5.
call "xsm_iset" using which, newval giving value.
```

DESCRIPTION

JAM has a number of global parameters and settings. This function is used to modify the current value of integer globals. The variable to change is specified by `which`. The new value is specified by `newval`. If you wish to get the value of a global integer use `xsm_inquire`.

The permissible values for the argument `which` are defined in the header file `smglobs.incl.cobol`. The following values are available:

Value	*Quantity*	*Meaning*
I-NODISP	0	Disable updating of display.
	1	Enable updating of display.
I-INSMODE	0	Enter overtype mode.
	1	Enter insert mode.

RETURNS

If `which` is one of the permissible values, the former value of the appropriate variable is returned.
1 True, the flag was set to on.
0 False, the flag was set to off.
–1 otherwise.

RELATED FUNCTIONS

```
call "xsm_finquire" using field-number, which giving value.
```

```
call "xsm_gp_inquire" using group-name, which giving value.
call "xsm_inquire" using which giving value.
call "xsm_pinquire" using which giving buffer.
call "xsm_pset" using which, newval giving buffer.
```

isselected

determine whether a radio button or checklist occurrence has been selected

SYNOPSIS

```
77 group-name          display-2 pic x(256).
77 group-occurrence    pic S(9)9 comp-5.
77 status              pic S(9)9 comp-5.
call "xsm_isselected" using group-name, group-occurrence giving
       status.
```

DESCRIPTION

This function lets you check to see whether or not a specific occurrence within a check list or radio button has been selected. The selection is referenced by the group name and occurrence number. If the occurrence is selected, `xsm_isselected` returns a `1`. A `0` is returned if the occurrence is not selected. See the Author's Guide for a more detailed discussion of groups.

Radio button and checklist occurrences are selected by using `xsm_select`. Using `xsm_select` on a radio button occurrence causes the current selection to be deselected. Checklist occurrences are deselected with `xsm_deselect`.

RETURNS

-1 arguments do not reference a checklist or radio button occurrence.
0 not selected.
1 selected.

RELATED FUNCTIONS

```
call "xsm_deselect" using group-name, group-occurrence giving
       status.
call "xsm_getfield" using buffer, field-number giving length.
call "xsm_intval" using field-number giving value.
call "xsm_select" using group-name, group-occurrence giving
       status.
```

issv

determine if a screen is in the saved list

SYNOPSIS

```
77 screen-name        display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_issv" using screen-name giving status.
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. This function searches the save list for a single screen and returns 1 is the screen is found (See `xsm_svscreen`).

This function is generally called by applications at screen entry to avoid re-acquiring data (via a database query) for previously saved screens. To accomplish this, first use `xsm_svscreen` to add the screen to the save list upon screen exit. Next, use `xsm_issv` to check the save list upon screen entry. If the screen is on the save list, you know that it has been previously displayed.

RETURNS

1 if the screen is in the saved list.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_svscreen" using screen-list, count giving status.
```

itofield

write an integer value to a field

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 value              pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_itofield" using field-number, value giving status.
```

DESCRIPTION

The integer passed to `xsm_itofield` is converted to characters and placed in the specified field. A number longer than the field will be truncated, on the left or right, according to the field's justification, without warning.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
call "xsm_e_itofield" using field-name, element, value giving
      status.
call "xsm_i_itofield" using field-name, occurrence, value
      giving status.
call "xsm_n_itofield" using field-name, value giving status.
call "xsm_o_itofield" using field-number, occurrence, value
      giving status.
```

RELATED FUNCTIONS

```
call "xsm_intval" using field-number giving value.
```

jclose

close current window or form under **JAM** Executive control

SYNOPSIS

```
77 status              pic S(9)9 comp-5.
call "xsm_jclose" giving status.
```

DESCRIPTION

The active screen is closed, and the display is restored to the state before the screen was opened. `xsm_jclose` should only be used when the **JAM** Executive is in use.

In the case of closing a form, `xsm_jclose` pops the form stack and calls `xsm_jform` to display the screen on the top of the form stack.

In the case of closing a window, `xsm_jclose` calls `xsm_close_window`. Since windows are stacked, the effect of closing a window is to return to the previous window. The cursor reappears at the same position it had before the window was opened.

RETURNS

-1 if there is no window open, i.e. if the currently displayed screen is a form (or if there is no screen displayed).
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_close_window" giving status.
call "xsm_jform" using screen-name giving status.
call "xsm_jwindow" using screen-name giving status.
```

jform

display a screen as a form under **JAM** control

SYNOPSIS

```
77 screen-name        display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_jform" using screen-name giving status.
```

DESCRIPTION

This function must be used with the JAM Executive. If you are not using the JAM Executive, use `xsm_r_form` or one of its variants. If you wish to display a window under JAM control, use `xsm_jwindow`.

This function displays the named screen as a form. You may close the form with `xsm_jclose`, or leave the task to the JAM Executive (e.g., when the user presses the EXIT key). Bringing up a screen as a form causes the previously displayed form and windows to be discarded, and their memory freed. The new form is placed on top of the JAM's form stack.

The difference between `xsm_jform` and `xsm_r_form`, other than the function arguments, is that only `xsm_jform` manipulates the form stack. Since `xsm_jform` calls `xsm_r_form`, refer to `xsm_r_form` for information on other details, such as how the screen to be displayed is found.

The character string `screen-name` uses the same format as that of a JAM control string that displays a form. In addition to the screen's name, you may optionally specify the position of the form on the physical display, the size of the viewport, and which portion of the form will be positioned in the viewport's top–left corner. See the Authoring Reference in the Author's Guide for details of viewport positioning. The following are all legal strings:

```
move "form" to screen-name.
call "xsm_jform" using screen-name giving status.
```

Display form's first row and column at the top–left corner of the physical display.

```
move "(20,10)form" to screen-name.
call "xsm_jform" using screen-name giving status.
```

Display form's first row and column at the 20th row and 10th column of the physical display.

```
move "(20,10,15,8)form" to screen-name.
call "xsm_jform" using screen-name giving status.
```

Display the first row and column of the form at the 20th row and 10th column of the physical display in viewport that is 15 rows by 8 columns.

A form may be larger than the viewport. If the viewport does not fit on the screen where indicated, **JAM** will attempt to place it entirely on the display at a different location. If you specify a viewport that is larger than the physical display, the viewport will be the size of the physical display. If you wish to change the viewport size after the window is displayed, use `xsm_viewport`.

RETURNS

0 if no error occurred.
–1 if the screen file's format is incorrect.
–2 if the screen cannot be found.
–4 if, after the display has been cleared, the screen cannot be successfully displayed because of a read error.
–5 if, after the display was cleared, the system ran out of memory.

RELATED FUNCTIONS

```
call "xsm_r_form" using screen-name giving status.
call "xsm_jwindow" using screen-name giving status.
```

jplcall

execute a JPL jpl procedure

SYNOPSIS

```
77 jplcall-text       display-2 pic x(256).
77 return-value       pic S(9)9 comp-5.
call "xsm_jplcall" using jplcall-text giving return-value.
```

DESCRIPTION

This function executes a JPL procedure precisely as if the following JPL statement were executed from within a JPL procedure:

```
jpl jplcall-text
```

For example, if the value of `jplcall-text` were:

```
verifysal :name 50000
```

then

and

```
jpl verifysal :name 50000
```

would be equivalent. See the JPL Programmer's Guide for further information on the JPL `jpl` command.

RETURNS

-1 if the procedure could not be loaded.
Otherwise, the value returned by the JPL procedure.

jplload

execute the JPL load command

SYNOPSIS

```
77 module-name-list   display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_jplload" using module-name-list giving status.
```

DESCRIPTION

This function is the COBOL interface to the JPL `load` command. Use this command to load one or more modules into memory.

The character string `module-name-list` may be one or more module names. Separate module names with a space.

Calling `xsm_jplload` has precisely the same effect as using the JPL `load` command. See the JPL Programmer's Guide for further information on the JPL `load` command.

Use `xsm_jplunload` to remove a module from memory.

RETURNS

-1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_jplpublic" using module-name-list giving status.
call "xsm_jplunload" using module-name giving status.
```

jplpublic

execute the JPL public command

SYNOPSIS

```
77 module-name-list   display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_jplpublic" using module-name-list giving status.
```

DESCRIPTION

This function is the COBOL interface to the JPL `public` command. Use this command to load one or more modules into memory.

The character string `module-name` may be one or more module names. Separate module names with a space.

Calling `xsm_jplpublic` has precisely the same effect as using the JPL `public` command. See the JPL Programmer's Guide for further information on the JPL `public` command.

Use `xsm_jplunload` to remove a module from memory.

RETURNS

-1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_jplload" using module-name-list giving status.
call "xsm_jplunload" using module-name giving status.
```

jplunload

execute the JPL unload command

SYNOPSIS

```
77 module-name        display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_jplunload" using module-name giving status.
```

DESCRIPTION

This function is the COBOL interface to the JPL `unload` command. Use this command to remove one or more modules from memory. Modules are read into memory by using either `xsm_jplpublic` or `xsm_jplload` or via the corresponding JPL commands.

Calling `xsm_jplunload` has precisely the same effect as using the JPL `unload` command. See the JPL Programmer's Guide for further information on the JPL `unload` command.

The character string `module-name` may be one or more module names. Separate module names with a space.

RETURNS

-1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_jplload" using module-name-list giving status.
call "xsm_jplpublic" using module-name-list giving status.
```

jtop
start the **JAM** Executive

SYNOPSIS

```
77 screen-name        display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_jtop" using screen-name giving status.
```

DESCRIPTION

All applications using the **JAM** Executive must include a call to `xsm_jtop`. This function starts the **JAM** Executive. The argument `screen-name` is the name of the first screen that your application displays. It will be displayed as a form. Once `xsm_jtop` is called the **JAM** Executive is in control until the user exits the application.

The **JAM** Executive makes calls to various **JAM** functions that handle all of the tasks needed to control the flow of an application such as opening the keyboard for input, opening and closing forms and windows, and processing all control strings.

If you do not use `xsm_jtop` you will have to write your own procedures to control the flow of your application. See the **JAM** Development Overview for a more detailed discussion of the **JAM** Executive.

RETURNS

0 Always.

jwindow

display a window at a given position under **JAM** control

SYNOPSIS

```
77 screen-name        display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_jwindow" using screen-name giving status.
```

DESCRIPTION

This function must be used with the JAM Executive. If you are not using the JAM Executive, use `xsm_r_window` or one of its variants. If you wish to display a form under JAM control, use `xsm_jform`.

This function displays the named screen as a window, by calling `xsm_r_window`. You may close the window with a call to `xsm_jclose`, or leave the task to the JAM Executive (e.g., when the user presses the EXIT key).

There is currently no difference between `xsm_jwindow` and `xsm_r_window` except for their arguments (although `xsm_jwindow` is not supported unless the JAM Executive is in use). See the description of `xsm_r_window` for the details of the behavior of `xsm_jwindow`.

The character string `screen-name` uses a format similar to that of a JAM control string that displays a window. Use a single ampersand to specify a stacked window and a double ampersand to specify a sibling window. If the ampersand is omitted, then the screen will be opened as a stacked window. In addition to the screen's name, you may optionally specify the position of the window on the physical display, the size of the viewport, as well as which portion of the window will be positioned in the viewport's top-left corner. The positioning and sizing syntax is identical to that of `xsm_jform`. See `xsm_jform` for examples of acceptable strings.

RETURNS

0 if no error occurred during display of the screen
-1 if the screen file's format is incorrect
-2 if the form cannot be found
-3 if the system ran out of memory but the previous screen was restored

RELATED FUNCTIONS

```
call "xsm_jclose" giving status.
call "xsm_jform" using screen-name giving status.
```

```
call "xsm_r_window" using screen-name, start-line, start-column
      giving status.
```

keyfilter

control keystroke record/playback filtering

SYNOPSIS

```
77 flag                pic S(9)9 comp-5.
77 old-flag            pic S(9)9 comp-5.
call "xsm_keyfilter" using flag giving old-flag.
```

DESCRIPTION

This function turns the keystroke record/playback mechanism of `xsm_getkey` on (`flag` = 1) or off (`flag` = 0). If no key recording or playback function has been installed, turning the mechanism on has no effect.

It returns a flag indicating whether recording was previously on or off.

RETURNS

The previous value of the filter flag.

RELATED FUNCTIONS

```
call "xsm_getkey" giving key.
```

keyhit

test whether a key has been typed ahead

SYNOPSIS

```
77 interval          pic S(9)9 comp-5.
77 status            pic S(9)9 comp-5.
call "xsm_keyhit" using interval giving status.
```

DESCRIPTION

This function checks whether a key has already been hit; if so, it returns 1 immediately. If not, it waits for the indicated interval and checks again. The key (if any is struck) is *not* read in, and is available to the usual keyboard input routines.

`interval` is in tenths of seconds; the exact length of the wait depends on the granularity of the system clock, and is hardware– and operating–system dependent. **JAM** uses this function to decide when to call the user–supplied asynchronous function.

If the operating system does not support reads with timeout, this function ignores the interval and only returns 1 if a key has been typed ahead.

RETURNS

0 if no key is available,
non–0 otherwise.

RELATED FUNCTIONS

```
call "xsm_getkey" giving key.
```

keyinit

initialize key translation table

SYNOPSIS

```
copy "keyfile.incl.cobol".

77 status              pic S(9)9 comp-5.
call "xsm_keyinit" using KEY-ADDRESS giving status.
```

DESCRIPTION

This routine is called by `xsm_initcrt` as part of the initialization process, but it can also be called by an application program (either before or after `xsm_initcrt`) to install a memory–resident key translation file.

`KEY-ADDRESS` is the address of a key translation table contained in `key-file.incl.cobol`, created using the `key2bin` and `bin2cob` utilities.

RETURNS

0 if the key file is successfully installed.
Program exit if the key file is invalid.

VARIANTS

```
call "xsm_n_keyinit" using key-file giving status.
```

keylabel

get the printable name of a logical key

SYNOPSIS

```
copy "smkeys.incl.cobol".

77 buffer              display-2 pic x(256).
77 key                 pic S(9)9 comp-5.
call "xsm_keylabel" using key giving buffer.
```

DESCRIPTION

Returns the label defined for `key` in the key translation file; the label is usually what is printed on the key on the physical keyboard. If there is no such label, returns the name of the logical key from the following table. Here is a list of key values:

NOTE: In the COBOL interface, each of the values listed in the table below should have the suffix `-KEY` added to it. So, for example, the value `EXIT` becomes `EXIT-KEY`.

Logical Key Values							
EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL	BACK
HOME	DELE	INS	LP	FERA	CLR	SPGU	SPGD
LSHF	RSHF	LARR	RARR	DARR	UARR	REFR	EMOH
INSL	DELL	ZOOM	SFTS	MTGL	VWPT	MOUS	
PF1-PF24		SPF1-SPF24		APP1-APP24		SFT1-SFT24	

If the key code is invalid (not one defined in `smkeys.incl.cobol`), this function returns an empty string.

RETURNS

A string naming the key, or an empty string if it has no name.

keyoption

set cursor control key options

SYNOPSIS

```
copy "smkeys.incl.cobol".

77 key                pic S(9)9 comp-5.
77 mode               pic S(9)9 comp-5.
77 newval             pic S(9)9 comp-5.
77 oldval             pic S(9)9 comp-5.
call "xsm_keyoption" using key, mode, newval giving oldval.
```

DESCRIPTION

Use `xsm_keyoption` to alter at run-time the behavior of `xsm_input` when a particular key is pressed. The default values for key options are built in to JAM. This function only works with cursor control keys. Cursor control keys include all JAM logical keys, *except* for PF, SPF, and APP keys. See "Key File" in the Configuration Guide.

There are three different possible values for `mode`: `KEY-ROUTING`, `KEY-GROUP`, and `KEY-XLATE`. The values that they use are defined in `smkeys.incl.cobol`. All of these modes draw on the following values for `key`.

NOTE: In the COBOL interface, each of the values listed in the table below should have the suffix `-KEY` added to it. So, for example, the value `EXIT` becomes `EXIT-KEY`.

Logical Key Values							
EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL	BACK
HOME	DELE	INS	LP	FERA	CLR	SPGU	SPGD
LSHF	RSHF	LARR	RARR	DARR	UARR	REFR	EMOH
INSL	DELL	ZOOM	SFTS	MTGL	VWPT	MOUS	

■`KEY-ROUTING`

Allows access to the EXECUTE and RETURN bits of the routing table. This mode is generally used to disable a key or to control explictly what action is taken when a key is hit. The following values may be assigned to `newval`:

1. KEY-IGNORE Disables key. JAM does nothing when key is struck.
2. EXECUTE The action normally associated with key is executed. May be summed with RETURN.
3. RETURN No action is performed, but the function returns to the caller in your code. Used to gain direct control of key's action. May be summed with EXECUTE.

■KEY-GROUP

Allows access to the group action bits. Use this function to control the action of the cursor when it is within a group. The following values may be assigned to newval:

1. VF-GROUP Obey group semantics. Hitting key will cause the cursor to move to the next field within the group in the indicated direction. If this value is summed with VF-CHANGE the cursor will exit the group in the indicated direction.
2. VF-CHANGE This value has no effect, unless it is summed with VF-GROUP. In this case the cursor will exit the group in the indicated direction.
3. 0 Assigning zero to newval will cause key to treat a field within a group as if it were not part of a group.
4. VF-OFFSCREEN Offscreen data will scroll onscreen from the direction indicated.
5. VF-NOPROT key will move cursor into a field protected from tabbing.

■KEY-XLATE

Allows access to the cursor table. Use this routine to assign key the action preformed by newval. newval may be any of the logical keys listed in the table above. This can often replace a user-supplied key change function.

RETURNS

-1 if some parameter is out of range.
the old value otherwise.

keyset

open a keyset

SYNOPSIS

```
copy "smsoftk.incl.cobol".

77 name                display-2 pic x(256).
77 scope               pic S(9)9 comp-5.
77 status              pic S(9)9 comp-5.
call "xsm_r_keyset" using name, scope giving status.


copy "mykeyset.incl.cobol".

77 scope               pic S(9)9 comp-5.
77 status              pic S(9)9 comp-5.
call "xsm_d_keyset" using ADDRESS, scope giving status.
```

DESCRIPTION

Use `xsm_d_keyset` and `xsm_r_keyset` to display a keyset. The parameter `name` is the name of the keyset. `scope` must be one of the values listed in `smsoftk.incl.cobol`. Application programs will normally use scope `KS-APPLIC`. Values for `scope` are defined in `smsoftk.incl.cobol`. For a more detailed explanation of scope see the Key Set chapter of the Author's Guide.

If there is currently a keyset of the specified scope the name of that keyset is compared with the name passed. If they are the same the present routine returns immediately. This means that if you want to "refresh" a keyset with a new copy from disk, you must first close the keyset with a call to `xsm_c_keyset`.

If the call is not successful then the current keyset remains displayed and an error message is posted to the end–user, except where noted otherwise.

The most commonly used variant is `xsm_r_keyset`. You do not need to know where the keyset resides because `xsm_r_keyset` searches for you. It looks first in the memory resident form list, next in any open libraries, then on disk in the directory specified by the argument to `xsm_initcrt`, and finally in the directories specified by `SMPATH`. Keyset files may be mixed freely with screen files in the screen list and in libraries.

You may save processing time by using `xsm_d_keyset` to display a memory–resident keyset. `ADDRESS` is the address of the keyset contained in the file `mykeyset.incl.cobol`. Use the utility `bin2cob` to create program data structures, from disk–based keysets, that you can compile into your application.

To close a keyset use `xsm_c_keyset`.

RETURNS

0 If no error occurred during display of the keyset.
–1 If the format incorrect (not a keyset).
–2 if the keyset cannot be found. No message is posted to the end–user.
–3 If the terminal doesn't support soft keys (or scope out of range).
–4 If there is a read error.
–5 If there is a malloc failure.

kscscope

query current keyset scope

SYNOPSIS

```
copy "smsoftk.incl.cobol".

77 scope              pic S(9)9 comp-5.
call "xsm_kscscope" giving scope.
```

DESCRIPTION

This routine returns the scope of the current keyset or `-1` if no keyset is currently active.

This function can be used to determine whether or not the application keyset (as opposed to the system keyset) is currently displayed.

Values for `scope` are defined in `smsoftk.incl.cobol`.

RETURNS

Current scope, or
–1 if not found.

RELATED FUNCTIONS

```
call "xsm_ksinq" using scope, number-keys, number-rows,
       current-row, maximum-len, keyset-name giving status.
call "xsm_skvinq" using scope, value, occurrence, attribute,
       label1, label2 giving status.
```

ksinq

inquire about keyset information

SYNOPSIS

```
copy "smsoftk.incl.cobol".

77 scope              pic S(9)9 comp-5.
77 number-keys        pic S(9)9 comp-5.
77 number-rows        pic S(9)9 comp-5.
77 current-row        pic S(9)9 comp-5.
77 maximum-len        pic S(9)9 comp-5.
77 keyset-name        display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_ksinq" using scope, number-keys, number-rows,
       current-row, maximum-len, keyset-name giving status.
```

DESCRIPTION

Use this routine to obtain the name, number of rows, number of items within a row, and current row of a keyset currently in memory. You supply the keyset's `scope` and five addresses to hold the information returned by `xsm_skinq`. `scope` must be one of the values defined in `smsoftk.incl.cobol`.

The function places the number of rows in the keyset in `number-row`, the number of soft keys per row in `number-keys`, and the current row number in `current-row`. The name of the keyset is placed in the pre-allocated buffer `keyset-name`. The size of `keyset-name` is specified by `maximum-len`. If the name of the keyset in longer then `keyset-name`, then `xsm_ksinq` fills the buffer to the end without adding a null character, otherwise a null character is added to the end of the string. The null pointer may be used for any or all of the parameters about which you do not desire information.

RETURNS

0 if information is returned.
-1 if there is no active keyset for the given scope.
-2 for an invalid scope.

RELATED FUNCTIONS

```
call "xsm_kscscope" giving scope.
call "xsm_skinq" using scope, row, softkey, value,
       display-attribute, label1, label2 giving status.
```

```
call "xsm_skvinq" using scope, value, occurrence, attribute,
      label1, label2 giving status.
```

ksoff

turn off soft key labels

SYNOPSIS

```
call "xsm_ksoff".
```

DESCRIPTION

When a keyset is opened with any of the library routines, the labels are automatically displayed. If you do not wish to display the labels at any point within your application, use `xsm_ksoff` to turn the display off.

If you wish to turn them the label display back on, use `xsm_kson`.

RELATED FUNCTIONS

```
call "xsm_kson".
```

kson

turn on soft key labels

SYNOPSIS

```
call "xsm_kson".
```

DESCRIPTION

Normally, keyset labels are displayed when a keyset is called up. The only way the display can be turned off is with the library routine, `xsm_ksoff`. Use this routine to turn the label display back on.

RELATED FUNCTIONS

```
call "xsm_ksoff".
```

l_close

close a library

SYNOPSIS

```
77 lib-desc          pic S(9)9 comp-5.
77 status            pic S(9)9 comp-5.
call "xsm_l_close" using lib-desc giving status.
```

DESCRIPTION

Closes the library indicated by `lib-desc` and frees all associated memory. The library descriptor is a number returned by a previous call to `xsm_l_open`.

RETURNS

–1 is returned if the library file could not be closed.
–2 is returned if the library was not open.
0 is returned if the library was closed successfully.

RELATED FUNCTIONS

```
call "xsm_l_at_cur" using lib-desc, screen-name giving status.
call "xsm_l_form" using lib-desc, screen-name giving status.
call "xsm_l_open" using lib-name giving lib-desc.
call "xsm_l_window" using lib-desc, screen-name, start-line,
       start-column giving status.
```

l_open
open a library

SYNOPSIS

```
77 lib-name          display-2 pic x(256).
77 lib-desc          pic S(9)9 comp-5.
call "xsm_l_open" using lib-name giving lib-desc.
```

DESCRIPTION

You must use `xsm_l_open` to open a library before you use a JPL module, a keyset, or a screen that is stored in the library. Use the utility `formlib` to create a library. (See the JAM Utilites Guide).

This routine allocates space in which to store information about the library, leaves the library file open, and returns a descriptor identifying the library. The descriptor may subsequently be used by `xsm_l_window` and related functions, to display screens stored in the library. The library can also be referenced implicitly by `xsm_r_window`, `xsm_r_keyset`, and `xsm_jplcall`, as well as related functions, which search all open libraries.

The library file is sought in all the directories identified by `SMPATH` and the parameter to `xsm_initcrt`. If you define the `SMFLIBS` variable in your setup file as a list of library names `xsm_l_open` will automatically be called for those libraries. The `xsm_r_` routines will then search in the specified libraries.

Several libraries may be kept open at once. This may cause problems on systems with severe limits on memory or simultaneously open files.

RETURNS

–1 if the library cannot be opened or read.
–2 if too many libraries are already open.
–3 if the named file is not a library.
–4 if insufficient memory is available.
Otherwise, a non–negative integer that identifies the library file.

RELATED FUNCTIONS

```
call "xsm_jplcall" using jplcall-text giving return-value.
call "xsm_jplload" using module-name-list giving status.
call "xsm_jplpublic" using module-name-list giving status.
call "xsm_l_at_cur" using lib-desc, screen-name giving status.
```

```
call "xsm_l_close" using lib-desc giving status.
call "xsm_l_form" using lib-desc, screen-name giving status.
call "xsm_l_window" using lib-desc, screen-name, start-line,
       start-column giving status.
call "xsm_r_at_cur" using screen-name giving status.
call "xsm_r_form" using screen-name giving status.
call "xsm_r_keyset" using name, scope giving status.
call "xsm_r_window" using screen-name, start-line, start-column
       giving status.
```

last

position the cursor in the last field

SYNOPSIS

```
call "xsm_last".
```

DESCRIPTION

Use this function to place the cursor at the first enterable position of the last tab–unprotected field of the current screen. If the last field unprotected from tabbing is right justified, the cursor is placed in the rightmost position of the field. By the same token, if the last unprotected field is left justified, the cursor is placed in the leftmost position of the field.

Unlike `xsm_home`, `xsm_last` will not reposition the cursor if the screen has no unprotected fields.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

This function is called when the JAM logical key EMOH is struck.

RELATED FUNCTIONS

```
call "xsm_backtab".
call "xsm_home" giving field-number.
call "xsm_nl".
call "xsm_tab".
```

lclear

erase LDB entries of one scope

SYNOPSIS

```
77 scope                pic S(9)9 comp-5.
77 status               pic S(9)9 comp-5.
call "xsm_lclear" using scope giving status.
```

DESCRIPTION

This function erases the values stored in the local data block for all names having a scope of the argument `scope`. Legal values for scope are between `1` and `9`. Constant variables having scope `1` *can* be erased.

Refer to the LDB chapter of the Programmer's Guide for a discussion of the scope of LDB entries.

RETURNS

–1 if scope is invalid.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_lreset" using file-name, scope giving status.
```

ldb_init

initialize (or reinitialize) the local data block

SYNOPSIS

```
call "xsm_ldb_init".
```

DESCRIPTION

This function creates an empty index of named data items by reading the data dictionary, then loads values into them from initialization files. Data Dictionary entries with a scope of 0 are not loaded into the LDB. There is no LDB prior to the first execution of this function.

Selected parts of the LDB, namely those assigned a certain scope, can be reinitialized using `xsm_lclear` or `xsm_lreset`.

This function is called explicitly in `jmain.cobol` and `jxmain.cobol`. Other functions that affect its behavior, such as `xsm_dicname` and `xsm_ininames`, should be called first.

RELATED FUNCTIONS

```
call "xsm_dicname" using dic-name giving status.
call "xsm_ininames" using name-list giving status.
call "xsm_lreset" using file-name, scope giving status.
```

leave

prepare to leave a **JAM** application temporarily

SYNOPSIS

```
call "xsm_leave".
```

DESCRIPTION

At times it may be necessary to leave a **JAM** application temporarily. For example you may need to escape to the command interpreter or to execute some graphics functions. In such a case, the terminal and its operating system channel need to be restored to their normal states.

This function should be called before leaving. It clears the physical screen (but not the internal screen image); resets the operating system channel; and resets the terminal (using the RESET sequence found in the video file).

RELATED FUNCTIONS

```
call "xsm_return".
```

length

get the maximum length of a field

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 field-length       pic S(9)9 comp-5.
call "xsm_length" using field-number giving field-length.
```

DESCRIPTION

This function returns the maximum length of the field specified by `field-number`. If the field is shiftable, its maximum shifting length is returned. This length is as defined in the JAM Screen Editor, and has no relation to the current contents of the field. Use `xsm_dlength` to get the length of the contents.

RETURNS

Length of the field.
0 if the field is not found.

VARIANTS

```
call "xsm_n_length" using field-name giving field-length.
```

RELATED FUNCTIONS

```
call "xsm_dlength" using field-number giving data-length.
```

lngval

get the long integer value of a field

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 value              pic S9(9) comp-5.
call "xsm_lngval" using field-number giving value.
```

DESCRIPTION

This function returns the contents of `field-number`, converted to a long integer. All non–digit characters are ignored, except for a leading plus or minus sign.

RETURNS

The long value of the field.
0 if the field is not found.

VARIANTS

```
call "xsm_e_lngval" using field-name, element giving value.
call "xsm_i_lngval" using field-name, occurrence giving value.
call "xsm_n_lngval" using field-name giving value.
call "xsm_o_lngval" using field-number, occurrence giving
      value.
```

RELATED FUNCTIONS

```
call "xsm_intval" using field-number giving value.
call "xsm_ltofield" using field-number, value giving status.
```

lreset

reinitialize LDB entries of one scope

SYNOPSIS

```
77 file-name          display-2 pic x(256).
77 scope              pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_lreset" using file-name, scope giving status.
```

DESCRIPTION

This function sets local data block entries to values read from `file-name`. The `scope` must be between 1 and 9. References in the file to LDB entries not belonging to `scope` are ignored. All variables belonging to `scope` are cleared before reinitializing. This means that `xsm_lreset` erases variables that are not in the file.

The file may be in the current directory, or in any of the directories listed in the SMPATH environment variable. It contains pairs of names with values, each enclosed in quotes. While all whites space outside the quotes is ignored, we recommend for readability that the file have one name–value pair per line. If an entry has multiple occurrences, it may be subscripted in the file. Here are a few sample pairs:

```
"husband"   "Ronald Reagan"
"wife[1]"   "Jane Wyman"
"wife[2]"   "Nancy Davis"
```

If you plan to use this function, we recommend that you group your variables in separate files by scope. You can use `xsm_ininames` to list a number of files for initialization.

RETURNS

–1 if file not found or scope out of range.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_lclear" using scope giving status.
```

lstore

copy everything from screen to LDB

SYNOPSIS

```
77 status              pic S(9)9 comp-5.
call "xsm_lstore" giving status.
```

DESCRIPTION

This function copies data from the screen to local data block entries with matching names.

The JAM Executive automatically calls `xsm_lstore` when bringing up a new screen or before closing a window. This function need not be called by application code except under special circumstances.

RETURNS

–3 if sufficient memory is not available.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_allget" using respect-flag.
```

ltofield

place a long integer in a field

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 value              pic S9(9) comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_ltofield" using field-number, value giving status.
```

DESCRIPTION

The long integer passed to this routine is converted to human–readable form and placed in `field-number`. If the number is longer than the field, it is truncated without warning, on the right or left depending on the field's justification.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
call "xsm_e_ltofield" using field-name, element, value giving
      status.
call "xsm_i_ltofield" using field-name, occurrence, value
      giving status.
call "xsm_n_ltofield" using field-name, value giving status.
call "xsm_o_ltofield" using field-number, occurrence, value
      giving status.
```

RELATED FUNCTIONS

```
call "xsm_itofield" using field-number, value giving status.
call "xsm_lngval" using field-number giving value.
```

m_flush

flush the message line

SYNOPSIS

```
call "xsm_m_flush".
```

DESCRIPTION

This function forces updates to the message line to be written to the display. This is useful if you want to display the status of an operation with `xsm_d_msg_line`, without flushing the entire display as `xsm_flush` does.

RELATED FUNCTIONS

```
call "xsm_flush".
```

max_occur

get the maximum number of occurrences

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 maximum            pic S(9)9 comp-5.
call "xsm_max_occur" using field-number giving maximum.
```

DESCRIPTION

This function returns the maximum number of occurrences that the array can hold as defined in the JAM Screen Editor or by `xsm_sc_max`. If you wish to find out the highest occurrence number of an array that actually contains data, use `xsm_num_occurs`.

RETURNS

0 if the field designation is invalid.
1 for a non–scrollable single field.
The number of elements in a non–scrollable array.
The maximum number of occurrences in a scrollable array.

VARIANTS

```
call "xsm_n_max_occur" using field-name giving maximum.
```

RELATED FUNCTIONS

```
call "xsm_num_occurs" using field-number giving number.
```

mnutogl

switch between menu mode and data entry mode on a dual–purpose screen

SYNOPSIS

```
copy "smumisc.incl.cobol".
77 screen-mode        pic S(9)9 comp-5.
77 old-mode           pic S(9)9 comp-5.
call "xsm_mnutogl" using screen-mode giving old-mode.
```

DESCRIPTION

JAM supports the use of a single screen as both a menu and a data entry screen, but the screen must be in one or the other "mode" at any given moment. This function can be used to change the mode of the screen and to test which mode the screen is in currently. The `mode` argument may have one of four values as defined in `smumisc.incl.cobol`:

Value	*Meaning*
`IN-AUTO`	No action (generally used just to test the return value).
`IN-DATA`	Change the screen to data entry mode.
`IN-MENU`	Change the screen to menu mode.
`IN-TOGL`	Toggle the screen from one mode to the other (akin to the `MTGL` logical key).

This function is similar to the built–in control function `jm_mnutogl`.

RETURNS

The mode that the screen was in before the function was called (`IN-DATA` or `IN-MENU`.)
–1 if the mode specification is invalid.

msg

display a message at a given column on the status line

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 column              pic S(9)9 comp-5.
77 disp-length         pic S(9)9 comp-5.
77 text                display-2 pic x(256).
call "xsm_msg" using column, disp-length, text.
```

DESCRIPTION

The message is merged with the current contents of the status line, and displayed beginning at `column`. `disp-length` gives the number of characters to display.

On terminals with onscreen attributes, the column position may need to be adjusted to allow for attributes embedded in the status line. Refer to `xsm_d_msg_line` for an explanation of how to embed attributes and function key names in a status line message.

This function is called by the function that updates the cursor position display (see `xsm_c_vis`).

RELATED FUNCTIONS

```
call "xsm_d_msg_line" using message, display-attribute.
```

msg_get

find a message given its number

SYNOPSIS

```
copy "smerror.incl.cobol".

77 buffer            display-2 pic x(256).
77 number            pic S(9)9 comp-5.
call "xsm_msg_get" using number giving buffer.
```

DESCRIPTION

The messages used by JAM library routines are stored in binary message files, which are created from text files using the JAM utility, `msg2bin`. Use `xsm_msgread` to load message files for use by this function.

This function takes the `number` of the message desired and returns the message, or a less informative string if the message number cannot be matched.

Messages are divided into classes based on their numbers, with up to 4096 messages per class. The message class is the message number divided by 4096, and the message offset within the class is the remainder. Predefined JAM message numbers and classes are shown in Appendix B..

RETURNS

The desired message, if found
otherwise, the message class and number, as a string

RELATED FUNCTIONS

```
call "xsm_msgfind" using number giving buffer.
call "xsm_msgread" using code, class, mode, arg giving status.
```

msgfind

find a message given its number

SYNOPSIS

```
copy "smerror.incl.cobol".

77 buffer              display-2 pic x(256).
77 number              pic S(9)9 comp-5.
call "xsm_msgfind" using number giving buffer.
```

DESCRIPTION

This function takes the `number` of a Screen Manager message, and returns the message string. It is identical to `xsm_msg_get`, except that it returns zero if the message number is not found.

Screen Manager message numbers and classes are shown in Appendix B..

RETURNS

The message
0 if the message number is out of range

RELATED FUNCTIONS

```
call "xsm_msg_get" using number giving buffer.
call "xsm_msgread" using code, class, mode, arg giving status.
```

msgread

read message file into memory

SYNOPSIS

```
copy "msgfile.incl.cobol".

copy "smerror.incl.cobol".

77 code              display-2 pic x(256).
77 class             pic S(9)9 comp-5.
77 mode              pic S(9)9 comp-5.
77 arg               display-2 pic x(256).
77 status            pic S(9)9 comp-5.
call "xsm_msgread" using code, class, mode, arg giving status.
```

DESCRIPTION

Reads a single set of messages from a binary message file into memory, after which they can be accessed using `xsm_msg_get` and `xsm_msgfind`. The `code` argument selects a single message class from a file that may contain several classes:

Code	*Class*	*Message Type*
SM	SM-MSGS	Screen Manager
FM	FM-MSGS	Screen Editor
JM	JM-MSGS	JAM run-time
JX	JX-MSGS	Data Dictionary & Control Strings
UT	UT-MSG	Utilities
(blank)		Undesignated user

`class` identifies a class of messages. Classes 0–7 are reserved for user messages, and several classes are reserved to JAM; see `smerror.incl.cobol`. As messages with the prefix `code` are read from the file, they are assigned numbers sequentially beginning at 4096 times `class`.

`mode` is a value composed from the following list. The first five indicate where to get the message file; at least one of these must be supplied. The latter four modify the basic action.

Value	*Action*
MSG-DELETE	Delete the message class and recover its memory.
MSG-DEFAULT	Use the default file defined by the setup variable SMMSGS.
MSG-FILENAME	Use the file named by arg.
MSG-ENVIRON	Use the file named in an environment variable named by arg.
MSG-MEMORY	Use a memory-resident file whose address is given by arg.
MSG-NOREPLACE	Modifier: do not overwrite previously installed messages.
MSG-DSK	Modifier: leave file open, do not read into memory
MSG-INIT	Modifier: do not use screen manager error reporting.
MSG-QUIET	Modifier: do not report errors.

You can sum MSG-NOREPLACE with any mode except MSG-DELETE, to prevent overwriting messages read previously. Error messages will be displayed on the status line, if the screen has been initialized by xsm_initcrt; otherwise, they will go to the standard error output. You can sum MSG-INIT with the mode to force error messages to standard error. Combining the mode with MSG-QUIET suppresses error reporting altogether.

If you sum MSG-DSK with the mode, the messages are not read into memory. Instead the file is left open, and xsm_msg_get and xsm_msgfind fetch them from disk when requested. If your message file is large, this can save substantial memory; but you should remember to account for operating system file buffers in your calculations.

arg contains the environment variable name for MSG-ENVIRON; the file name for MSG-FILENAME; or the address of the memory-resident file contained in msgfile.incl.cobol for MSG-MEMORY. It may be passed as zero for other modes.

RETURNS

0 if the operation completed successfully.
1 if the message class was already in memory and the mode included MSG-NOREPLACE.
2 if the mode was MSG-DELETE and the message file was not in memory.
-1 if the mode was MSG-ENVIRON and the environment variable was undefined.
-2 if the mode was MSG-ENVIRON or MSG-FILENAME and the message file could

not be read from disk; other negative values if the message file was bad or insufficient memory was available.

RELATED FUNCTIONS

```
call "xsm_msg_get" using number giving buffer.
call "xsm_msgfind" using number giving buffer.
```

mwindow

display a status message in a window

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 text              display-2 pic x(256).
77 line              pic S(9)9 comp-5.
77 column            pic S(9)9 comp-5.
77 status            pic S(9)9 comp-5.
call "xsm_mwindow" using text, line, column giving status.
```

DESCRIPTION

This function displays `text` in a pop–up window, whose upper left–hand corner appears at `line` and `column`. The line and column are counted from `0`. If `line` is `1`, the top of the window will be on the second line of the display. The window itself is constructed on the fly by the run–time system. No data entry is possible in it, nor is data entry possible in underlying screens as long as it is displayed.

Due to the delayed write feature in **JAM**, you should call `xsm_flush` to cause the screen to be updated and the message to be displayed, unless you call `xsm_input` directly after the call to `xsm_mwindow`. `xsm_close_window` may be used to close a window called with `xsm_mwindow`.

All the percent escapes for status messages, except **%M** and **%W**, are effective. Refer to `xsm_err_reset` for a list and full description. If either `line` or `column` is negative, the window will be displayed according to the rules given for `xsm_r_at_cur`.

RETURNS

–1 if there was a malloc failure.
1 if the text had to be truncated to fit in a window.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_d_msg_line" using message, display-attribute.
```

n_

variants that take a field name only

SYNOPSIS

```
77 field-name          display-2 pic x(256).
call "xsm_n_..." using field-name, ....
```

DESCRIPTION

The n_ functions access a field by means of the field/group name. For a complete description of individual functions, look under the related function without n_ in its name. For example, xsm_n_amt_format is described under xsm_amt_format. If the named field/group is not on the screen, these functions will attempt to access a similarly named entry in the local data block.

name

obtain field name given field number

SYNOPSIS

```
77 buffer              display-2 pic x(256).
77 field-number        pic S(9)9 comp-5.
call "xsm_name" using field-number giving buffer.
```

DESCRIPTION

Given a field number, `xsm_name` returns a buffer that contains the field name referenced by `field-number`.

RETURNS

The name of the field referenced, if found.
0 otherwise.

nl

position cursor to the first unprotected field beyond the current line

SYNOPSIS

```
call "xsm_nl".
```

DESCRIPTION

This function moves the cursor to the next occurrence of an array, scrolling if necessary. Unlike the down–arrow, it will allocate an empty scrolling occurrence if there are no more below but the maximum has not yet been exceeded.

If the current field is not scrolling, the cursor is positioned to the first unprotected field, if any, following the current *line* of the form. If there are no unprotected fields beyond the current field, the cursor is positioned to the first unprotected field of the screen.

If the screen has no unprotected fields at all, the cursor is positioned to the first column of the line following the current line. If the cursor is on the last line of the form, it goes to the top left–hand corner of the screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

This function is ordinarily bound to the RETURN key.

RELATED FUNCTIONS

```
call "xsm_backtab".
call "xsm_home" giving field-number.
call "xsm_last".
call "xsm_tab".
```

novalbit

forcibly invalidate a field

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_novalbit" using field-number giving status.
```

DESCRIPTION

Resets the VALIDED bit of the specified field, so that the field will again be subject to validation when it is next exited, or when the screen is validated as a whole.

JAM sets a field's VALIDED bit automatically when the field passes all its validations. The bit is initially clear, and is cleared whenever the field is altered by keyboard input or by a library function such as `xsm_putfield`.

RETURNS

-1 if the field is not found.
0 otherwise.

VARIANTS

```
call "xsm_e_novalbit" using field-name, element giving status.
call "xsm_i_novalbit" using field-name, occurrence giving
      status.
call "xsm_n_novalbit" using field-name giving status.
call "xsm_o_novalbit" using field-number, occurrence giving
      status.
```

RELATED FUNCTIONS

```
call "xsm_fval" using field-number giving status.
call "xsm_s_val" giving status.
```

null

test if field is null

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_null" using field-number giving status.
```

DESCRIPTION

Use `xsm_null` to test a field to see whether it has both the null edit and contains the null character string that has been assigned to that field. See null edits in the Author's Guide.

RETURNS

1 If the field has the null edit and contains the appropriate null character string.
–1 if the field does not exist.
0 otherwise.

VARIANTS

```
call "xsm_e_null" using field-name, element giving status.
call "xsm_i_null" using field-name, occurrence giving status.
call "xsm_n_null" using field-name giving status.
call "xsm_o_null" using field-number, occurrence giving status.
```

num_occurs

find the highest numbered occurrence containing data

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 number             pic S(9)9 comp-5.
call "xsm_num_occurs" using field-number giving number.
```

DESCRIPTION

This function returns the highest occurrence number of the array specified by `field-number` that actually contains data. The field number may be that of any field with the array.

Most of the time the highest numbered occurrence containing data will be the same as the number of occurrences actually containing data. However, it is possible to have blank occurrences preceding occurrences containing data.

This count is different from the maximum capacity of an array, which you can retrieve with `xsm_max_occur`.

RETURNS

The highest numbered occurrence containing data.
0 if there is no data in the field.
-1 if the field is not found.

VARIANTS

```
call "xsm_n_num_occurs" using field-name giving number.
```

o_

variants that take a field number and occurrence number

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 occurrence         pic S(9)9 comp-5.
call "xsm_o_..." using field-number, occurrence, ....
```

DESCRIPTION

The o_ functions refer to data by field number and occurrence number. An occurrence is a slot within an array of fields in which data may be stored. Occurrences may be either on or off-screen. Since **JAM** treats an individual field as an array with one field, even a single non-scrolling field is considered to have one occurrence. The **JAM** library contains routines that allow you to manipulate individual occurrences during run-time.

If the occurrence is zero, the reference is always to the current contents of the specified field.

For the description of a particular function, look under the related function without o_ in its name. For example, `xsm_o_amt_format` is described under `xsm_amt_format`.

occur_no

get the current occurrence number

SYNOPSIS

```
77 occurrence          pic S(9)9 comp-5.
call "xsm_occur_no" giving occurrence.
```

DESCRIPTION

This function returns the occurrence number of the field beneath the cursor. If the field is an element of a non–scrollable array, the occurrence number is the same as the field's element number. Likewise, the occurrence number of a single non–scrolling field is `1`.

RETURNS

0 if the cursor is not in a field.
Otherwise, the occurrence number.

RELATED FUNCTIONS

```
call "xsm_getcurno" giving field-number.
```

off_gofield

move the cursor into a field, offset from the left

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 offset             pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_off_gofield" using field-number, offset giving
      status.
```

DESCRIPTION

This function moves the cursor into `field-number`, at position `offset` within the field's contents, regardless of the field's justification. The field's contents will be shifted if necessary to bring the appropriate piece onscreen.

If `offset` is larger than the field length (or the maximum length if the field is shiftable), the cursor will be placed in the rightmost position.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
call "xsm_e_off_gofield" using field-name, element, offset
      giving status.
call "xsm_i_off_gofield" using field-name, occurrence, offset
      giving status.
call "xsm_n_off_gofield" using field-name, offset giving
      status.
call "xsm_o_off_gofield" using field-number, occurrence, offset
      giving status.
```

RELATED FUNCTIONS

```
call "xsm_disp_off" giving offset.
call "xsm_gofield" using field-number giving status.
call "xsm_sh_off" giving offset.
```

option

set a Screen Manager option

SYNOPSIS

```
copy "smmisc.incl.cobol".
copy "smmisc2.incl.cobol".
copy "smmisc3.incl.cobol".

77 option              pic S(9)9 comp-5.
77 newval              pic S(9)9 comp-5.
77 oldval              pic S(9)9 comp-5.
call "xsm_option" using option, newval giving oldval.
```

DESCRIPTION

Use `xsm_option` to alter during run-time the default Screen Manager options defined in the files `smmisc.incl.cobol`, `smmisc2.incl.cobol` and `smmisc3.incl.cobol`. Possible options include, error window attributes, delayed write options, cursor display and zoom options. It is only necessary to include those header files for the settings you wish to change. See the "Setup File" section in the *Configuration Guide* for a list of options and possible values. Use `xsm_keyoption` to alter the behavior of cursor control keys.

If you wish to simply inquire as to an option's current value, use the value NOCHANGE (defined in `smsetup.incl.cobol`) for `newval`.

This function replaces the following version 4.0 functions: `xsm_ch_emsgatt`, `xsm_ch_form_atts`, `xsm_ch_qmsgatt`, `xsm_ch_umsgatt`, `xsm_dw_options`, `xsm_er_options`, `xsm_fcase`, `xsm_fextension`, `xsm_ind_set`, `xsm_mp_options`, `xsm_mp_string`, `xsm_ok_options`, `xsm_stextatt`, and `xsm_zm_options`. They are included in your version 5.0 library only for backward compatibility. We strongly recommend that you do not use them in the future.

RETURNS

The old value for the specified option.
-1 if the option is out of range.

RELATED FUNCTIONS

```
call "xsm_keyoption" using key, mode, newval giving oldval.
```

oshift

shift a field by a given amount

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 offset             pic S(9)9 comp-5.
77 return-value       pic S(9)9 comp-5.
call "xsm_oshift" using field-number, offset giving
       return-value.
```

DESCRIPTION

This function shifts the contents of `field-number` by `offset` positions. If `offset` is negative, the contents are shifted right (data past the left-hand edge of the field become visible); otherwise, the contents are shifted left. Shifting indicators, if displayed, are adjusted accordingly.

The field may be shifted by fewer than `offset` positions if the maximum shifting width is reached with less shifting.

RETURNS

The number of positions actually shifted.
0 if the field is not found or is not shifting.

VARIANTS

```
call "xsm_n_oshift" using field-name, offset giving
       return-value.
```

pinquire

obtain value of a global strings

SYNOPSIS

```
copy "smglobs.incl.cobol".

77 buffer              display-2 pic x(256).
77 which               pic S(9)9 comp-5.
call "xsm_pinquire" using which giving buffer.
```

DESCRIPTION

This function is used to obtain the current value of a global pointer variable. The values for `which` are defined in `smglobs.incl.cobol`. If you wish to modify a global string use `xsm_pset`.

Pointer values for `which` are defined in `smglobs.incl.cobol`. They are:

Value	*Meaning*
`P-YES`	The `Y` character for YES/NO field. This is returned as a three character string. The first character is the lowercase yes value, the second character is the uppercase yes value, and the third character is the null terminator.
`P-NO`	The `N` character for YES/NO field. This is returned as a three character string. The first character is the lowercase no value, the second character is the uppercase no value, and the third character is the null terminator.
`P-DECIMAL`	This is returned as a three character string. The first character is the user's decimal point marker, the second character is the operating system's decimal point marker, and the third character is the null terminator.
`P-FLDPTRS`	Pointer to an array of field structures. The implementation of these structures is very release dependent.
`P-TERM`	Returns the name **JAM** uses as the terminal identifier or the null string if not found.
`P-SPMASK`	Pointer to an memory–resident full size form containing all blanks.

Value	*Meaning*
P-USER	Pointer to developer-specified region of memory. This pointer is not set by **JAM**; it is set and maintained, if desired, by the application.
SP-NAME	Name of the active screen.
SP-STATLINE	Text of current status line.
SP-STATATTR	Attributes of current status line (pointer to array of unsigned short integers).
P-DICNAME	Name of data dictionary file.
V-	Any of the "V-" values defined in `smvideo.incl.cobol` may be passed to obtain various video related information.

In general, the objects pointed to by the pointers returned by `xsm_pinquire` have limited duration and should be used or copied quickly (except for P-USER, which is maintained by the application). The P- pointers point to the actual objects within **JAM**. The SP- pointers point to copies of the objects. Since the characteristics of these objects are implementation dependent, they may change in future releases of **JAM**. In no case (except P-USER) should the objects be modified directly through the pointers returned by `xsm_pinquire`. Use `xsm_pset` to modify selected objects).

RETURNS

If the argument corresponds to a global pointer variable, the value of that variable is returned.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_finquire" using field-number, which giving value.
call "xsm_gp_inquire" using group-name, which giving value.
call "xsm_iset" using which, newval giving value.
call "xsm_pset" using which, newval giving buffer.
```

protect

protect an array

SYNOPSIS

```
copy "smvalids.incl.cobol".


77 field-number       pic S(9)9 comp-5.
77 mask                pic S(9)9 comp-5.
77 status              pic S(9)9 comp-5.

call "xsm_aprotect" using field-number, mask giving status.
call "xsm_aunprotect" using field-number, mask giving status.
call "xsm_protect" using field-number giving status.
call "xsm_unprotect" using field-number giving status.
call "xsm_lprotect" using field-number, mask giving status.
call "xsm_lunprotect" using field-number, mask giving status.
```

DESCRIPTION

There are four types of protection associated with fields and arrays, any combination of which may be assigned: data entry, tabbing into, clearing, and validation. `xsm_protect` and `xsm_unprotect` always set and clear all four types of protection. The remaining protection functions set and clear any combination of protection, as specified by `mask`. The values for `mask` are defined in `smvalids.incl.cobol` and are listed below. Combinations may be specified by adding values together.

Value for mask	*Meaning*
EPROTECT–FEDIT	protect from data entry
TPROTECT–FEDIT	protect from tabbing into and from entering via any other key
CPROTECT–FEDIT	protect from clearing
VPROTECT–FEDIT	protect from validation
ALLPROTECT–FEDIT	protect from all of the above

Protection is associated an individual field (i.e. an element), and with an array as a whole. Therefore, all offscreen array occurrences always share the same level of protection,

while the onscreen occurrences have the levels of protection (possibly all different) associated with their host fields (i.e. elements). Since protection is associated with individual fields, and not with individual occurrences, deleting an occurrence with `xsm_doccur` will not scroll up the protection with the occurrences.

`xsm_protect`, `xsm_unprotect`, `xsm_1protect`, and `xsm_1unprotect` set and clear protection for individual fields. `xsm_aprotect` and `xsm_aunprotect` set and clear protection for all of the fields of an array, and for the array as a whole (the `field-number` may specify any field in the array). For example, unprotecting an array with `xsm_aunprotect` will undo protection done by `xsm_1protect`. A subsequent call to `xsm_1protect` will re-protect the specified field of the array, but can never affect the offscreen occurrences of the array.

Caution: It is generally safer to protect and unprotect arrays with `xsm_aprotect` and `xsm_aunprotect`, rather than with the field-oriented protection functions.

RETURNS

-1 if the field does not exist;
0 otherwise.

VARIANTS

```
call "xsm_n_protect" using field-name giving status.
call "xsm_e_protect" using field-name, element giving status.
call "xsm_n_unprotect" using field-name giving status.
call "xsm_e_unprotect" using field-name, element giving status.
call "xsm_n_1protect" using field-name, mask giving status.
call "xsm_e_1protect" using field-name, element, mask giving
     status.
call "xsm_n_1unprotect" using field-name, mask giving status.
call "xsm_e_1unprotect" using field-name, element, mask giving
     status.
call "xsm_n_aprotect" using field-name, mask giving status.
call "xsm_n_aunprotect" using field-name, mask giving status.
```

pset

Modify value of global strings

SYNOPSIS

```
copy "smglobs.incl.cobol".

77 buffer              display-2 pic x(256).
77 which               pic S(9)9 comp-5.
77 newval              display-2 pic x(256).
call "xsm_pset" using which, newval giving buffer.
```

DESCRIPTION

This function is used to modify the contents of a global string. The string you wish to change is specified by `which`. The value that you wish to change the variable to is specified by `newval`. If you wish only to get the value of a global string use `xsm_pinquire`.

The following values for `which`, defined in `smglobs.incl.cobol`, are available:

Value	*Meaning*
`P-YES`	The `Y` character for YES/NO field. This is specified by a three character string. The first character is the lowercase yes value, the second character is the uppercase yes value, and the third character is the null terminator.
`P-NO`	The `N` character for YES/NO field. This is specified by a three character string. The first character is the lowercase no value, the second character is the uppercase no value, and the third character is the null terminator.
`P-DECIMAL`	This is specified by a three character string. The first character is the user's decimal point marker, the second character is the operating system's decimal point marker, and the third character is the null terminator.

RETURNS

If `which` is one of the above, the old contents of the corresponding array are returned. 0 otherwise.

RELATED FUNCTIONS

```
call "xsm_iset" using which, newval giving value.
```

```
call "xsm_pinquire" using which giving buffer.
```

putfield

put a string into a field

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 data               display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_putfield" using field-number, data giving status.
```

DESCRIPTION

The string `data` is moved into the field specified by `field-number`. Strings that are too long will be truncated without warning, while strings shorter than the destination field are blank filled (to the left if the field is right justified, otherwise to the right). If `data` is a null string, then the field is cleared. This causes date and time fields that take system values to be refreshed.

This function sets the field's MDT bit to indicate that it has been modified, and clears its VALIDED bit to indicate that the field must be revalidated upon exit. `xsm_n_putfield` and `xsm_i_putfield` will store `data` in the LDB if the named field is not present in the screen. However, if the LDB item has a scope of 1 (constant), its contents will be unaltered and the function will return –1.

In variants that take `name` as an argument, `name` can be either the name of a field or a group. In the case of a group, the functions `xsm_select` and `xsm_deselect` should be used to change the group's value.

Notice that the order of arguments to this function is different from that of arguments to the related function `xsm_getfield`.

RETURNS

–1 if the field is not found; 0 otherwise.

VARIANTS

```
call "xsm_e_putfield" using name, element, data giving status.
call "xsm_i_putfield" using name, occurrence, data giving
      status.
call "xsm_n_putfield" using name, data giving status.
call "xsm_o_putfield" using field-number, occurrence, data
      giving status.
```

RELATED FUNCTIONS

```
call "xsm_deselect" using group-name, group-occurrence giving
      status.
```

```
call "xsm_getfield" using buffer, field-number giving length.
call "xsm_select" using group-name, group-occurrence giving
      status.
```

putjctrl

associate a control string with a key

SYNOPSIS

```
copy "smkeys.incl.cobol".

77 key                pic S(9)9 comp-5.
77 control-string     display-2 pic x(256).
77 default            pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_putjctrl" using key, control-string, default giving
       status.
```

DESCRIPTION

Each JAM screen contains a table of control strings associated with function keys. JAM also maintains a default table of keys and control strings, which take effect when the current screen has no control string for a function key you press. This table enables you to define system-wide actions for keys. It is initialized from SMINICTRL setup variables. See the section on setup in the Configuration Guide for further information.

This function associates `control-string` with `key` in one of the tables, replacing the control string previously associated with `key` (if there was one). If `default` is zero, the control string will be installed in the current screen, and will disappear when you exit the screen; otherwise, it will go into the system-wide default table. If `control-string` is empty, the existing control string, if any, will be deleted. If both screen and default control strings exist for a given key, deleting the control string for the screen will put the default control string into effect.

If you install a default control string for a key that is defined in the current screen, the definition in the screen will be used. Note also that JAM will not search the form or window stack for function key definitions; only the current screen and the default table are consulted. Values for `key` are in `smkeys.incl.cobol`. The syntax for control strings is defined in the Author's Guide.

RETURNS

-5 if insufficient memory is available; 0 otherwise.

pwrap

put text to a wordwrap field

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 text               display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_pwrap" using field-number, text giving status.
```

DESCRIPTION

This function copies `text` to a wordwrap field specified by `field-number`. Wraps occur at the end of words. The last character of every line is a space. If a word is longer than one less than the length of the field, the word is broken one character short of the end of the field, a space is appended, and the remainder of the word wraps to the next line.

The variant `xsm_o_pwrap` copies the text into an array beginning at the specified occurrence.

Warning: If you attempt to copy data that is too large for the wordwrap field to hold, `xsm_pwrap` will truncate the excess text.

RETURNS

–1 if the field number is invalid.
–2 if the text was truncated because it was too long for the field.
0 otherwise.

VARIANTS

```
call "xsm_o_pwrap" using field-number, occurrence, text giving
      status.
```

RELATED FUNCTIONS

```
call "xsm_gwrap" using buffer, field-number, buffer-length
      giving length.
```

query_msg

display a question, and return a yes or no answer

SYNOPSIS

```
77 message            display-2 pic x(256).
77 reply              pic S(9)9 comp-5.
call "xsm_query_msg" using message giving reply.
```

DESCRIPTION

The `message` is displayed on the status line, until you type a yes or a no key. A yes key is the first letter of the `SM-YES` entry in the message file (or the XMIT key), and a no key is the first letter of the `SM-NO` entry (or the EXIT key); case is ignored. At that point, this function returns the lower case letter as defined in the message file to its caller.

All keys other than yes and no keys are ignored.

RETURNS

Lower-case ASCII 'y' or 'n', according to the response.

RELATED FUNCTIONS

```
call "xsm_d_msg_line" using message, display-attribute.
call "xsm_is_no" using field-number giving status.
call "xsm_is_yes" using field-number giving status.
```

qui_msg

display a message preceded by a constant tag, and reset the message line

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 message            display-2 pic x(256).
call "xsm_qui_msg" using message.
```

DESCRIPTION

This function prepends a tag (normally "ERROR:") to `message`, and displays the whole on the status line (or in a window if it is too long). The tag may be altered by changing the `SM-ERROR` entry in the message file. The message remains visible until the operator presses a key. Refer to the description of setup in the Configuration Guide for an exact description of error message acknowledgement. If the message is longer than the status line, it will be displayed in a window instead. If the cursor position display has been turned on (see `xsm_c_vis`), the end of the status line will contain the cursor's current row and column. If the message text would overlap that area of the status line, it will be displayed in a window instead.

This function is identical to `xsm_quiet_err`, except that it does not turn the cursor on. It is similar to `xsm_emsg`, which does not prepend a tag.

Several *percent escapes* provide control over the content and presentation of status messages. See `xsm_emsg` for details.

RELATED FUNCTIONS

```
call "xsm_emsg" using message.
call "xsm_err_reset" using message.
call "xsm_option" using option, newval giving oldval.
call "xsm_quiet_err" using message.
```

quiet_err

display error message preceded by a constant tag, and reset the status line

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 message           display-2 pic x(256).
call "xsm_quiet_err" using message.
```

DESCRIPTION

This function prepends a tag (normally "ERROR") to `message`, turns the cursor on, and displays the whole message on the status line (or in a window if it is too long). This function is identical to `xsm_qui_msg`, except that it turns the cursor on. It is similar to `xsm_err_reset`, which does not prepend a tag. Refer to `xsm_emsg` for an explanation of how to change display attributes and insert function key names within a message.

RELATED FUNCTIONS

```
call "xsm_emsg" using message.
call "xsm_err_reset" using message.
call "xsm_option" using option, newval giving oldval.
call "xsm_qui_msg" using message.
```

rd_part

read part of a record to the current screen

SYNOPSIS

```
copy "screen.jam.incl.cobol".

77 first-field        pic S(9)9 comp-5.
77 last-field         pic S(9)9 comp-5.
call "xsm_rd_part" using SCREEN, first-field, last-field.
```

DESCRIPTION

This function copies data from a record to all fields between `first-field` and `last-field` within the current screen, converting individual items as appropriate. An array and its scrolling occurrences will be copied only if the *first* element falls between `first-field` and `last-field`. This routine is commonly used with `xsm_wrt_part`, which writes part of the screen to a record. If you wish to read information into the entire screen, use `xsm_rdstruct`. To read information from a record defined in the data dictionary, use `xsm_rrecord`. Use `xsm_putfield` to write a string to an individual field.

The record SCREEN, contained in the file `screen.jam.incl.cobol` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gCOBOL screen.jam
```

Each item in the record is a field of the type specifed in the Screen Editor. If you specify the type `omit`, data will not be written into the field. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other record declaration. You can ignore the items that represent fields which do not fall within the bounds of the specifed fields. However, the record definition must contain all of the fields on the screen.

The arguments that represent the range of fields to be copied, `first-field` and `last-field` are passed as field numbers.

The record may be initialized with `xsm_wrt_part` or with data from elsewhere. Record items within the specified range which will not be initialized prior to calling `xsm_rd_part` must be cleared or you risk crashing your application when garbage is read into the screen.

Remember, you must update the record declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
call "xsm_putfield" using field-number, data giving status.
call "xsm_rd_struct" using screen, byte-count.
call "xsm_rrecord" using RECORD, record-name, byte-count.
call "xsm_wrt_part" using SCREEN, first-field, last-field.
```

rdstruct

read data from a record to the screen

SYNOPSIS

```
copy "screen.jam.incl.cobol".

77 byte-count          pic S(9)9 comp-5.
call "xsm_rd_struct" using SCREEN, byte-count.
```

DESCRIPTION

This function copies data from a record to the current screen, converting individual items as appropriate. It is commonly used with `xsm_wrtstruct`, which writes data from fields on the current screen to a record. If you wish to read information into a group of consecutively numbered fields, use `xsm_rd_part`. To read information from a record defined in the data dictionary, use `xsm_rrecord`. Use `xsm_putfield` to write a string to an individual field.

The record SCREEN, contained in the file `screen.jam.incl.cobol` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gCOBOL screen.jam
```

Each item in the record is a field of the type specifed in the Screen Editor. If you specify the type `omit`, data will not be written into the field. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

The argument `byte-count` is an integer variable. `xsm_rdstruct` will store in `byte-count` the number of bytes copied from the record.

The record may be initialized with `xsm_wrtstruct` or with data from elsewhere. items within the record that will not be initialized prior to calling `xsm_rdstruct` must be cleared or you risk crashing your application when garbage is read into the screen.

Remember, you must update the record declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
call "xsm_putfield" using field-number, data giving status.
call "xsm_rd_part" using SCREEN, first-field, last-field.
call "xsm_rrecord" using RECORD, record-name, byte-count.
call "xsm_wrtstruct" using SCREEN, byte-count.
```

rescreen

refresh the data displayed on the screen

SYNOPSIS

```
call "xsm_rescreen".
```

DESCRIPTION

This function repaints the entire display from JAM's internal screen and attribute buffers. Anything written to the screen by means other than JAM library functions will be erased. This function is normally bound to the RESCREEN key and executed automatically within `xsm_getkey`.

You may need to use this function after doing screen I/O with the flag `xsm_do_not_display` turned on, or after escaping from an JAM application to another program (see `xsm_leave`). If all you want is to force writes to the display, use `xsm_flush`.

RELATED FUNCTIONS

```
call "xsm_flush".
call "xsm_return".
```

resetcrt

reset the terminal to operating system default state

SYNOPSIS

```
call "xsm_resetcrt".
call "xsm_jresetcrt".
call "xsm_jxresetcrt".
```

DESCRIPTION

The function `xsm_resetcrt` is generally used only when you are writing your own Executive. It resets terminal characteristics to the operating system's normal state. Be sure to call `xsm_resetcrt` be called when leaving the Screen Manager environment (before program exit).

All the memory associated with the display and open screens is freed. However, the buffers holding the message file, key translation file, etc. are not released. A subsequent call to `xsm_initcrt` will find them in place. Then `xsm_resetcrt` clears the screen and turns on the cursor, transmits the `RESET` sequence defined in the video file, and resets the operating system channel.

The **JAM** Executive calls `xsm_resetcrt` via `xsm_jresetcrt` (or via `xsm_jxresetcrt` in the case of an authoring executable) automatically as part of its exit processing. It should not be called by application programs except in case of abnormal termination.

RELATED FUNCTIONS

```
call "xsm_cancel".
call "xsm_leave".
```

resize

notify **JAM** of a change in the display size

SYNOPSIS

```
77 rows                pic S(9)9 comp-5.
77 columns             pic S(9)9 comp-5.
77 status              pic S(9)9 comp-5.
call "xsm_resize" using rows, columns giving status.
```

DESCRIPTION

This function enables you to change the size of the display used by **JAM** from the default defined by the LINES and COLMS entries in the video file. It makes it possible to use a single video file in a windowing environment. Applications can be run in different sized windows with each application setting its display size at run time. It can also be used for switching between normal and compressed modes (e.g. 80 and 132 columns on VT100-compatible terminals).

If the specified rectangle is larger than the physical display, the results will be unpredictable. You may specify at most 255 rows or columns.

This function clears the physical and logical screens; any displayed forms or windows, together with data entered on them, are lost.

RETURNS

-1 if a parameter was less than 0 or greater than 255.
0 if successful.
Program exit on memory allocation failure.

return

prepare for return to **JAM** application

SYNOPSIS

```
call "xsm_return".
```

DESCRIPTION

This routine should be called upon returning to a JAM application after a temporary exit.

It sets up the operating system channel and initializes the display using the `SETUP` string from the video file. It does *not* restore the screen to the state it was in before `xsm_leave` was called. Use `xsm_rescreen` to accomplish that, if desired.

RELATED FUNCTIONS

```
call "xsm_leave".
call "xsm_resetcrt".
```

rmformlist

empty the memory–resident form list

SYNOPSIS

```
call "xsm_rmformlist.
```

DESCRIPTION

This function erases the memory–resident form list established by `xsm_formlist`, and releases the memory used to hold it. It does not release any of the memory–resident JPL modules, key sets, or screens themselves. Calling this function will prevent `xsm_r_window`, `xsm_r_keyset`, `xsm_jplcall`, and related functions from finding memory–resident objects.

RELATED FUNCTIONS

```
call "xsm_formlist" using name, address giving status.
```

rrecord

read data from a record defined in the data dictionary

SYNOPSIS

```
copy "record.incl.cobol".

77 record-name        display-2 pic x(256).
77 byte-count         pic S(9)9 comp-5.
call "xsm_rrecord" using RECORD, record-name, byte-count.
```

DESCRIPTION

This function reads data from a record into fields on the current screen. If a field is not on the current screen then the data is written to the LDB. This routine is commonly used with `xsm_wrecord`, which writes data from the screen and LDB to a record defined in the data dictionary. If you wish to read data into all of the fields within the current screen, use `xsm_rdstruct`. To copy data to a group of consecutively numbered fields, use `xsm_rd_part`. Use `xsm_putfield` to write a string to an individual field.

The record named `RECORD`, contained in the file `record.incl.cobol` can be created from the data dictionary file `data.dic` via the `dd2struct` utility as follows:

```
dd2struct -gCOBOL data.dic
```

Each record item is a field of the type specified in the Data Dictionary Editor. Data will be written into the field onscreen even if the `omit` type is specified. See "Data Type" in the Author's Guide and `dd2struct` in the Utilities Guide for further information.

Once created, the record declarations may be treated exactly like any other record declarations. The argument `record-name` is the name of the data dictionary entry from which the record was created.

The argument `byte-count` is an integer. Upon return from `xsm_rrecord`, the value contained in the integer will be the number of bytes or characters read from the record. The value will be 0 if an error occurred.

The record may be initialized with `xsm_wrecord` or with data from elsewhere. Items within the record that will not be initialized prior to calling `xsm_rrecord` must be cleared or you risk crashing your application when garbage is read into the screen or the LDB.

Remember, you must update the record declaration whenever you alter the data dictionary from which it was generated.

RELATED FUNCTIONS

```
call "xsm_putfield" using field-number, data giving status.
call "xsm_rd_part" using SCREEN, first-field, last-field.
call "xsm_rd_struct" using screen, byte-count.
call "xsm_wrecord" using RECORD, record-name, byte-count.
```

rscroll

scroll an array

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 req-scroll         pic S(9)9 comp-5.
77 lines              pic S(9)9 comp-5.
call "xsm_rscroll" using field-number, req-scroll giving lines.
```

DESCRIPTION

This function scrolls an array along with any synchronized arrays by `req-scroll` occurrences. If `req-scroll` is positive, the array scrolls down (towards the bottom of the data); otherwise, it scrolls up.

The function returns the actual amount scrolled. This could be the amount requested, or a smaller value if the requested amount would bring the array past its beginning or end. If 0 is returned it means that the array was at its beginning or end, or an error occurred. Negative numbers indicate scrolling up occurred.

RETURNS

The actual amount scrolled. Positive numbers indicate downward scrolling while negative numbers mean upward scrolling.
0 if no scrolling or error.

VARIANTS

```
call "xsm_n_rscroll" using field-name, req-scroll giving lines.
```

RELATED FUNCTIONS

```
call "xsm_ascroll" using field-number, occurrence giving
      status.
call "xsm_t_scroll" using field-number giving status.
```

s_val

validate the current screen

SYNOPSIS

```
77 status              pic S(9)9 comp-5.
call "xsm_s_val" giving status.
```

DESCRIPTION

This function validates each field and occurrence, whether on or offscreen, that is not protected from validation (VPROTECT). It is called automatically from `xsm_input` when the TRANSMIT key is hit while in data entry mode. `xsm_sval` also validates groups.

When the first element of a scrolling array is encountered, earlier offscreen occurrences are validated first. When the last element of a scrolling array is encountered, later offscreen occurrences are validated immediately after that element.

If synchronized arrays exist, the following occurs. When an offscreen occurrence is validated, the corresponding occurrences from synchronized arrays are validated as well. Synchronized array are validated in order according to their base field number. The offscreen occurrences *preceding* the synchronized arrays are validated before the first onscreen occurrence of the first (lowest base field number) of the synchronized arrays. Similarly, the offscreen occurrences *following* the arrays are validated immediately after the last onscreen occurrence of the last (highest base field number) array.

Validation	*Skip if valid*	*Skip if empty*
required	y	n
must fill	y	y
regular expression	y	y
range	y	y
check–digit	y	y
date or time	y	y
table lookup	y	y
currency format	y	n*
math expresssion	n	n

Validation	*Skip if valid*	*Skip if empty*
field validation	n	n
JPL function	n	n

* The currency format edit contains a skip–if–empty flag; see the Author's Guide.

If you need to force a skip–if–empty validation, make the field required. A field with embedded punctuation must contain at least one non–blank non–punctuation character in order to be considered non–empty; otherwise any non blank character makes the field non–empty.

If an occurrence fails validation, the cursor is positioned to it and an error message displayed. If the occurrence was offscreen, its the array is first scrolled to bring it onscreen. This routine returns at the first error; any fields past will not be validated.

RETURNS

–1 if any field fails validation.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_fval" using field-number giving status.
```

sc_max

alter the maximum number of occurrences allowed in a scrollable array

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 new-max            pic S(9)9 comp-5.
77 actual-max         pic S(9)9 comp-5.
call "xsm_sc_max" using field-number, new-max giving
       actual-max.
```

DESCRIPTION

This function changes the maximum number of occurrences allowed in `field-number`, and in all synchronized arrays. The original maximum is set when the screen is created. If the desired new maximum is less than the highest numbered occurrence that contains data, the new maximum will be set to the number of that occurrence (i.e., the value returned by `xsm_num_occurs`). The maximum can decrease only to a value between the highest numbered occurrence containing data and the previous maximum. It can never be less than the number of elements in the array.

RETURNS

The actual new maximum (see above).
0 if the desired maximum is invalid, or if the array is not scrollable.

VARIANTS

```
call "xsm_n_sc_max" using field-name, new-max giving
       actual-max.
```

RELATED FUNCTIONS

```
call "xsm_max_occur" using field-number giving maximum.
call "xsm_num_occurs" using field-number giving number.
```

sdtime

get formatted system date and time

SYNOPSIS

```
77 buffer              display-2 pic x(256).
77 format              display-2 pic x(256).
call "xsm_sdtime" using format giving buffer.
```

DESCRIPTION

This function gets the current date and/or time from the operating system and returns it in the form specified by `format`.

`format` is a string beginning with `y` or `n` followed by any combination of date/time tokens and literal text. `y` indicates a 12–hour clock; `n` (or any other character) indicates a 24–hour clock. This character must be given, even if the format does not include time tokens. The tokens are described in the table below. These tokens are case–sensitive.

Unit	*Description*	*Token*
Year	4 digit (e.g., 1990)	`%4y`
	2 digit (e.g., 90)	`%2y`
Month	1 or 2 digit (1 – 12)	`%m`
	2 digit (01 – 12)	`%0m`
	full name (e.g., January)	`%*m`
	3 character name (e.g., Jan)	`%3m`
Day	1 or 2 digit (1 – 31)	`%d`
	2 digit (01 – 31)	`%0d`
Day of the Week	full name (e.g. Sunday)	`%*d`
	3 character name (e.g., Sun)	`%3d`
Day of the Year	digit (1 – 365)	`%+d`

Unit	*Description*	*Token*
Hour	1 or 2 digit (1 – 12 or 1 – 24)	`%h`
	2 digit (01 –12 or 01 –24)	`%0h`
Minute	1 or 2 digit (1 – 59)	`%M`
	2 digit (01 – 59)	`%0M`
Second	1 or 2 digit (1 – 59)	`%s`
	2 digit (01 – 59)	`%0s`
AM or PM	for use with a 12–hour clock	`%p`
Literal Percent	use `%` as a literal character	`%%`
Ten Default Formats	`SM-0DEF-DTIME`	`%0f`
(from the message file)	`SM-1DEF-DTIME`	`%1f`
	...	`...`
	`SM-9DEF-DTIME`	`%09f`

At runtime, JAM strips off the first character of `format`. If the character is `y`, it uses a 12–hour clock; else it uses the default 24–hour clock. Next it examines the rest of `format`, replacing any tokens with the appropriate values. All other characters are used literally. Therefore, be sure to put a `y` or an `n` (or perhaps a blank) at the beginning of `format`. If you do not, JAM strips off the first token's percent sign and it treats the rest of the token as literal text.

You may also retrieve a date/time format from a field using `xsm_edit_ptr`.

The text for day and month names, AM and PM, as well as the tokens for the ten default formats, are all stored in the message file. These entries may be modified. See the Configuration Guide for details.

Note: This function replaces Release 4's `xsm_sdate` and `xsm_stime` function.

RETURNS

The current date/time in the specified format.
Empty if `format` is invalid.

RELATED FUNCTIONS

```
call "xsm_calc" using field-number, occurrence, expression
      giving status.
```

select

select a checklist or radio button occurrence

SYNOPSIS

```
77 group-name          display-2 pic x(256).
77 group-occurrence    pic S(9)9 comp-5.
77 status              pic S(9)9 comp-5.
call "xsm_select" using group-name, group-occurrence giving
       status.
```

DESCRIPTION

This function allows you to select a specific occurrence within a checklist or radio button. The group name and occurrence number are used to reference the desired selection.

Use `xsm_deselect` to deselect a checklist occurrence.

Selecting a radio button occurrence automatically causes the currently selected radio button to be deselected, because exactly one occurrence in a radio button group must be selected at all times. See the Author's Guide for a more detailed discussion of groups.

Use `xsm_isselected` to check whether or not a particular radio button or checklist occurrence is currently selected.

RETURNS

-1 arguments do not reference a checklist or radio button occurrence.
0 occurrence not previously selected.
1 occurrence previously selected.

RELATED FUNCTIONS

```
call "xsm_deselect" using group-name, group-occurrence giving
       status.
call "xsm_isselected" using group-name, group-occurrence giving
       status.
```

setbkstat

set background text for status line

SYNOPSIS

```
copy "smattrib.incl.cobol".

77 message           display-2 pic x(256).
77 display-attribute pic S(9)9 comp-5.
call "xsm_setbkstat" using message, display-attribute.
```

DESCRIPTION

The `message` is saved, to be shown on the status line whenever there is no higher priority message to be displayed. The highest priority messages are those passed to `xsm_d_msg_line`, `xsm_err_reset`, `xsm_quiet_err`, or `xsm_query_msg`; the next highest are those attached to a field by means of the status text option (see the **JAM** Author's Guide). Background status text has lowest priority.

Possible values for the `display-attribute` argument are defined in the header file `smattrib.incl.cobol`, as shown in the table below:

Foreground Attributes	*Background Attributes*
ATT–BLANK	ATT–BHILIGHT
ATT–REVERSE	
ATT–UNDERLN	
ATT–BLINK	
ATT–HILIGHT	
ATT–STANDOUT	
ATT–DIM	
ATT–ACS (alternate character set)	
Foreground Colors	*Background Colors*
ATT–BLACK	ATT–BBLACK
ATT–BLUE	ATT–BBLUE
ATT–GREEN	ATT–BGREEN

Foreground Colors	*Background Colors*
ATT-CYAN	ATT-BCYAN
ATT-RED	ATT-BRED
ATT-MAGENTA	ATT-BMAGENTA
ATT-YELLOW	ATT-BYELLOW
ATT-WHITE	ATT-BWHITE

Foreground colors may be used alone or added together with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting the foreground value will cause the attribute to default to black.

`xsm_setstatus` sets the background status to an alternating ready/wait flag; you should turn that feature off before calling this routine.

Refer to `xsm_d_msg_line` for an explanation of how to embed attribute changes and function key names into your message.

RELATED FUNCTIONS

```
call "xsm_d_msg_line" using message, display-attribute.
call "xsm_setstatus" using mode.
```

setstatus

turn alternating background status message on or off

SYNOPSIS

```
77 mode                pic S(9)9 comp-5.
call "xsm_setstatus" using mode.
```

DESCRIPTION

If mode is non–zero, alternating status flags are turned on. After this call, one message (normally Ready) is displayed on the status line while **JAM** is waiting for input, and another (normally Wait) when it is not. If mode is zero, the messages are turned off.

The status flags will be replaced temporarily by messages passed to xsm_err_reset or a related routine. They will overwrite messages posted with xsm_d_msg_line or xsm_setbkstat.

The alternating messages are stored in the message file as SM-READY and SM-WAIT, and can be changed there. Attribute changes and function key names can be embedded in the messages; refer to xsm_d_msg_line for instructions.

RELATED FUNCTIONS

```
call "xsm_setbkstat" using message, display-attribute.
```

sh_off

determine the cursor location relative to the start of a shifting field

SYNOPSIS

```
77 offset              pic S(9)9 comp-5.
call "xsm_sh_off" giving offset.
```

DESCRIPTION

Returns the difference between the start of data in a shiftable field and the current cursor location. If the current field is not shiftable, it returns the difference between the leftmost column of the field and the current cursor location, like `xsm_disp_off`.

RETURNS

The difference between the current cursor position and the start of shiftable data in the current field.
-1 if the cursor is not in a field.

RELATED FUNCTIONS

```
call "xsm_disp_off" giving offset.
```

shrink_to_fit

remove trailing empty array elements and shrink screen

SYNOPSIS

```
call "xsm_shrink_to_fit".
```

DESCRIPTION

Use this routine to dynamically downsize the current screen when you don't know how many elements of an array are going to be populated with data at run time. This routine removes all trailing elements in all arrays on screen and then shrinks the screen to a size just large enough to accommodate the displayed data. If no data is placed in the array, the entire array will be removed. Only the currently displayed copy of the screen in memory is altered.

This routine only downsizes the array and screen. It will not enlarge an array or screen that is too small to hold the information, so be sure to create, within the Screen Editor, an array and screen that can hold the largest amount of data that you plan on inserting.

sibling

define the current window as being or not being a sibling window

SYNOPSIS

```
77 should-it-be       pic S(9)9 comp-5.
call "xsm_sibling" using should-it-be.
```

DESCRIPTION

Users may switch between the active window and all siblings of that window while they are in viewport mode. Sibling windows must be next to each other on the window stack. When a window is defined as a sibling, then it and the window immediately beneath it on the window stack are considered to be siblings of one another. The user enters viewport mode when either the VWPT (viewport) logical key is pressed or when the application program makes a call to `xsm_winsize`.

Use this function to define whether or not the current window is defined as sibling. To change the current sibling status of a window assign `should-it-be` to:

0	No, it is not a sibling window.
1	Yes, it is a sibling window.

To understand how sibling windows work, imagine you have a stack of three windows: `window-top`, `window-middle`, and `window-bottom`. To make `window-top` and `window-middle` siblings of each other, define `window-top` as a sibling window. They are now considered siblings of each other. You can then add a third sibling to the pair, by defining `window-middle` as a sibling window. This results in `window-middle` and `window-bottom` becoming siblings of one another and consequently, `window-top` and `window-bottom` are also siblings of each other. There is no limit to the number of siblings window you may chain together in this fashion, as long as the windows are adjacent to each other on the stack.

If you wish to bring a different window to the top of the stack, use `xsm_wselect`. To get the number of windows currently in the window stack use `xsm_wcount`.

The base form can be a sibling of the windows adjacent to it.

RELATED FUNCTIONS

```
call "xsm_wcount" giving return-value.
call "xsm_winsize" giving status.
```

```
call "xsm_wselect" using window-number giving return-value.
```

size_of_array

get the number of elements

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 size               pic S(9)9 comp-5.
call "xsm_size_of_array" using field-number giving size.
```

DESCRIPTION

This function returns the number of elements in the array containing `field-number`. Elements are the onscreen portion of an array. An array always has at least one element.

RETURNS

0 if the field designation is invalid.
1 if the field is not an array.
The number of elements in the array otherwise.

VARIANTS

```
call "xsm_n_size_of_array" using field-name giving size.
```

RELATED FUNCTIONS

```
call "xsm_max_occur" using field-number giving maximum.
```

skinq

obtain soft key information by position

SYNOPSIS

```
copy "smsoftk.incl.cobol".

copy "smattrib.incl.cobol".

copy "smkeys.incl.cobol".

77 scope              pic S(9)9 comp-5.
77 row                pic S(9)9 comp-5.
77 softkey            pic S(9)9 comp-5.
77 value              pic S(9)9 comp-5.
77 display-attribute  pic S(9)9 comp-5.
77 label1             display-2 pic x(256).
77 label2             display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_skinq" using scope, row, softkey, value,
      display-attribute, label1, label2 giving status.
```

DESCRIPTION

Use this routine to obtain the value, attributes, and label of a soft key contained in a keyset currently in memory, given a soft key's position within a keyset.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Values for `scope` are defined in `smsoftk.incl.cobol`. For a more detailed explanation of scope see the Keyset chapter of the Programmer's Guide.

The logical value of the specified soft key is placed in `value`. This will be a number that corresponds to a value defined in `smkeys.incl.cobol`. A value of 0 means the key is inactive.

The attributes (color, blinking etc...) of the label will be placed in `display-attribute`. The attribute should be one of the values listed in `smattrib.incl.cobol`.

The first and second row labels are placed in `label1` and `label2` respectively. You should pre-allocate at least nine elements for `label1` and `label2` buffers (eight for the label characters and one for the null character).

If you do not desire information about one or more of these parameters you may assign the parameters the null pointer.

If you want general information about a keyset, see `xsm_ksinq`. If you want the scope of the current keyset, use `xsm_kscscope`.

WARNING: This routine can not be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs **JAM** automatically breaks the rows to position them correctly on the monitor. This means that you will not be able to reliably reference a particular soft key by its row and position. Instead, use `xsm_skvinq`.

RETURNS

0 if information has been returned.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
call "xsm_kscscope" giving scope.
call "xsm_ksinq" using scope, number-keys, number-rows,
       current-row, maximum-len, keyset-name giving status.
call "xsm_skvinq" using scope, value, occurrence, attribute,
       label1, label2 giving status.
```

skmark

mark or unmark a soft key label by position

SYNOPSIS

```
copy "smsoftk.incl.cobol".

77 scope              pic S(9)9 comp-5.
77 row                pic S(9)9 comp-5.
77 softkey            pic S(9)9 comp-5.
77 mark               pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_skmark" using scope, row, softkey, mark giving
       status.
```

DESCRIPTION

Use this routine to mark or unmark a soft key label in an open keyset. The mark is made in the last position of the first label.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.cobol`. The argument `row` is the row number in which the desired `softkey` resides. Rows are counted from top to bottom, beginning with `1`. The argument `softkey` is the position number within `row` of the desired soft key. Positions are numbered left to right, beginning with `1`.

The argument `mark` may be any single ASCII character. An asterisk (`*`) is the most commonly used mark. To unmark the key use the space character (' ') for `mark`.

The marking or unmarking of a soft key is often done to indicate a selection on a function key that toggles between two options.

WARNING: This routine can not be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs JAM automatically breaks the rows to position them correctly on the monitor. This means that you will not be able to reliably reference a particular soft key by its row and position. Instead, use `xsm_skvmark`.

RETURNS

0 if the marking was successful.
–1 if there is no keyset of the specified scope.
–2 if the scope is out of range.
–3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
call "xsm_skvmark" using scope, value, occurrence, mark giving
      status.
```

skset

set characteristics of a soft key by position

SYNOPSIS

```
copy "smsoftk.incl.cobol".

copy "smkeys.incl.cobol".

copy "smattrib.incl.cobol".

copy "smkeys.incl.cobol".

77 scope              pic S(9)9 comp-5.
77 row                pic S(9)9 comp-5.
77 softkey            pic S(9)9 comp-5.
77 value              pic S(9)9 comp-5.
77 attribute          pic S(9)9 comp-5.
77 label1             display-2 pic x(256).
77 label2             display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_skset" using scope, row, softkey, value, attribute,
      label1, label2 giving status.
```

DESCRIPTION

This routine can be used to modify a soft key's scope, value, attribute, or label of any currently open keysets. You may modify one or more of these specifications with each call of `xsm_skset`.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.cobol`. The argument `row` is the row number in which the desired `softkey` resides. Rows are counted from top to bottom, beginning with `1`. The argument `softkey` is the position number within `row` of the desired soft key. Positions are numbered left to right, beginning with `1`.

The `value` refers to the logical key name to be assigned to the soft key. Available values are defined in `smkeys.incl.cobol`. If you do not want to change the logical name, assign `-1` to `value`.

The `attribute` (color, blinking, etc.) is specified by using values listed in `smattrib.incl.cobol`. If you do not want to change `attribute`, assign it `0`. (Note: If

you set both the background and foreground to black, `xsm_skset` will set the foreground to white, provided that the terminal supports background color.)

The variables `label1` and `label2` are the first and second lines of the labels respectively. If you do not wish to change one of the labels, assign it the null pointer.

WARNING: This routine can not be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs **JAM** automatically breaks the rows to position them correctly on the monitor. This means that you will not be able to reliably reference a particular soft key by its row and position. Instead, use `xsm_skvset`.

RETURNS

0 if no error has occurred.
-1 if there is no active keyset for the given scope.
-2 for an invalid scope.
-3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
call "xsm_skvset" using scope, value, occurrence, newval,
       attribute, label1, label2 giving status.
```

skvinq

obtain soft key information by value

SYNOPSIS

```
copy "smsoftk.incl.cobol".

copy "smattrib.incl.cobol".

copy "smkeys.incl.cobol".

77 scope              pic S(9)9 comp-5.
77 value              pic S(9)9 comp-5.
77 occurrence         pic S(9)9 comp-5.
77 attribute          pic S(9)9 comp-5.
77 label1             display-2 pic x(256).
77 label2             display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_skvinq" using scope, value, occurrence, attribute,
       label1, label2 giving status.
```

DESCRIPTION

Use this routine to obtain the label text and attributes of a soft key contained in a keyset currently in memory, given the soft key's value. It can be used when the terminal has a different number of keys than the keyset was designed for.

The soft key is referenced by the keyset it belongs to, its value, and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.cobol`. The `value` of the soft key is one of the value defined in `smkeys.incl.cobol`. The argument `occurrence` specifies which occurrence of a key with the specified value is desired (in case of duplicates).

The attributes (color, blinking etc . . .) of the label will be placed in `attribute`. The attributes correspond to a value, or some combination of summed values listed in `smattrib.incl.cobol`.

The first and second row labels are placed in `label1` and `label2` respectively. You should pre–allocate at least nine elements for `label1` and `label2` buffers (eight for the label characters and one for the null character).

If you do not desire information about one or more of these parameters you may assign the parameters the null pointer.

For general information about a keyset, see `xsm_ksinq`. If you want the scope of the current keyset, use `xsm_kscscope`.

RETURNS

0 if information has been returned.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
call "xsm_skinq" using scope, row, softkey, value,
       display-attribute, label1, label2 giving status.
```

skvmark

mark a soft key by value

SYNOPSIS

```
copy "smsoftk.incl.cobol".

copy "smkeys.incl.cobol".

77 scope              pic S(9)9 comp-5.
77 value              pic S(9)9 comp-5.
77 occurrence         pic S(9)9 comp-5.
77 mark               pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_skvmark" using scope, value, occurrence, mark giving
       status.
```

DESCRIPTION

Use this routine to mark or unmark a soft key label in an open keyset. The mark is made in the last position of the first label.

The soft key is referenced by the keyset it belongs to, its value and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.cobol`. The `value` of the soft key is one of the value defined in `smkeys.incl.cobol`. The argument `occurrence` is the nth time that `value` appears in the keyset. If you wish to mark all occurrences of `value` assign 0 to `occurrence`.

The argument `mark` may be any single ASCII character. An asterisks (`*`) is the most commonly used mark. To unmark the key use the space character (' ') for `mark`.

The marking or unmarking of a soft key is often done to indicate a selection on a function key that toggles between two options.

RETURNS

0 if the mark was successful.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
call "xsm_skmark" using scope, row, softkey, mark giving
       status.
```

skvset

set characteristics of a soft key by value

SYNOPSIS

```
copy "smsoftk.incl.cobol".

copy "smattrib.incl.cobol".

copy "smkeys.incl.cobol".

77 scope              pic S(9)9 comp-5.
77 value              pic S(9)9 comp-5.
77 occurrence         pic S(9)9 comp-5.
77 newval             pic S(9)9 comp-5.
77 attribute          pic S(9)9 comp-5.
77 label1             display-2 pic x(256).
77 label2             display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_skvset" using scope, value, occurrence, newval,
       attribute, label1, label2 giving status.
```

DESCRIPTION

This routine can be used to modify the scope, value, attribute, or label of a soft key within a currently open keyset. You may modify one or more of these specifications with each call of `xsm_skset`.

The soft key is referenced by the keyset it belongs to, its value and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.cobol`. The `value` of the soft key is one of the value defined in `smkeys.incl.cobol`. The argument `occurrence` is the nth time that `value` appears in the keyset. If you wish to change all occurrences of `value` assign `0` to `occurrence`.

The value of `newvalue` refers to the logical key name to be assigned to the soft key. Available values are defined in `smkeys.incl.cobol`. If you do want to change the logical name, assign `-1` to `value`.

The `attribute` (color, blinking, etc.) is specified by using values listed in `smattrib.incl.cobol`. If you do not want to change `attribute`, assign it `0`. (Note: If you set both the background and foreground to black, `xsm_skset` will set the foreground to white, provided that the terminal supports background color.)

The variables `label1` and `label2` are the first and second lines of the labels respectively. If you do not wish to change one of the labels, assign it the null pointer.

RETURNS

0 if no error occurred
–1 if there is no active keyset for the given scope
–2 for an invalid scope
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
call "xsm_skset" using scope, row, softkey, value, attribute,
       label1, label2 giving status.
```

strip_amt_ptr

strip amount editing characters from a string

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 inbuf              display-2 pic x(256).
77 outbuf             display-2 pic x(256).
call "xsm_strip_amt_ptr" using field-number, inbuf giving
       outbuf.
```

DESCRIPTION

Strips all non-digit characters from the string, except for an optional leading minus sign and decimal point. If `inbuf` is not empty, `field-number` is ignored and the passed string is processed in place.

If `inbuf` is empty, the contents of `field-number` are used.

RETURNS

The stripped text,
0 if `inbuf` is empty and the field number is invalid.

RELATED FUNCTIONS

```
call "xsm_amt_format" using field-number, buffer giving status.
call "xsm_dblval" using field-number giving value.
```

submenu_close

close the current submenu

SYNOPSIS

```
77 status              pic S(9) 9 comp-5.
call "xsm_submenu_close" giving status.
```

DESCRIPTION

Submenus are ordinarily closed before `xsm_input` returns. It may, however, be told to leave them open by using the OK-LEAVEOPEN option, either in the setup file or via `xsm_option`. See the *Configuration Guide* for details. Regardless of how this option is set, submenus are automatically closed whenever the underlying window is closed with `xsm_close_window`.

This function, then, is needed only when all of the following conditions are true.

1. OK-LEAVEOPEN is in use.
2. The submenu is no longer needed.
3. Access is needed to the underlying window.

RETURNS

-1 if there is no submenu currently open.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_close_window" giving status.
```

svscreen

register a list of screens on the save list

SYNOPSIS

```
77 screen-list        display-2 pic x(256).
77 count              pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_svscreen" using screen-list, count giving status.
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. The number of screens to be added is given by `count`. You may add screens to the list anywhere within your code, however the screen is not actually placed in memory until it is closed for the first time. This means that the time saving factor only comes into play in subsequent openings of the screen. Any data entered into a screen will not be saved until the screen is closed.

Screens are removed from the list with `xsm_unsvscreen`. You can check to see if a screen is on the save list with `xsm_issv`. Checking the list prior to calling `xsm_svscreen`, however, is not crucial as any attempt to add a screen that is already on the list will have no effect.

This routine saves processing time at the expense of memory. It is best suited for use with screens that both require large amounts of data to be read in from elsewhere (databases, other files, etc.) and do not allow the user to enter data. For instance, if you have a help screen that needs to be populated by a data base and is going to be called up more then once, you can re–display the screen much more quickly by saving the screen in memory.

RETURNS

0 is returned if no error occurred.
1 is returned if registration failed (out of memory).

RELATED FUNCTIONS

```
call "xsm_issv" using screen-name giving status.
call "xsm_unsvscreen" using screen-list, count.
```

t_scroll

test whether an array can scroll

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_t_scroll" using field-number giving status.
```

DESCRIPTION

This function returns 1 if the array in question is scrollable, and 0 if not. The argument `field-number` may be any field within the array.

RETURNS

1 if the array is scrolling.
0 if it is not scrolling or if no such `field-number`.

RELATED FUNCTIONS

```
call "xsm_t_shift" using field-number giving status.
```

t_shift

test whether field can shift

SYNOPSIS

```
77 field-number       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_t_shift" using field-number giving status.
```

DESCRIPTION

This function returns 1 if the field in question is shiftable, and 0 if not or if there is no such field.

RETURNS

1 if field is shifting.
0 if not shifting or `field-number` is invalid.

RELATED FUNCTIONS

```
call "xsm_t_scroll" using field-number giving status.
```

tab

move the cursor to the next unprotected field

SYNOPSIS

```
call "xsm_tab".
```

DESCRIPTION

If the cursor is in a field with a next–field edit and one of the fields specified by the edit is unprotected from tabbing, the cursor is moved to the first enterable position of that field. Otherwise, the cursor is advanced to the first enterable position of the next tab unprotected field on the screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

RELATED FUNCTIONS

```
call "xsm_backtab".
call "xsm_home" giving field-number.
call "xsm_last".
call "xsm_nl".
```

tst_all_mdts

find first modified occurrence

SYNOPSIS

```
77 occurrence          pic S(9)9 comp-5.
77 field-number        pic S(9)9 comp-5.
call "xsm_tst_all_mdts" using occurrence giving field-number.
```

DESCRIPTION

This function tests the MDT bits of all occurrences of all fields on the current screen, and returns the base field and occurrence numbers of the first occurrence with its MDT set, if there is one. The MDT bit indicates that an occurrence has been modified, either from the keyboard or by the application program, since the screen was displayed (or since its MDT was last cleared by `xsm_bitop`).

This function returns zero if no occurrences have been modified. If one has been modified, it returns the base field number, and stores the occurrence number in `occurrence`.

RETURNS

0 if no MDT bit is set anywhere on the screen

The number of the first field on the current screen for which some occurrence has its MDT bit set. In this case, the number of the first occurrence with MDT set is returned in `occurrence`.

RELATED FUNCTIONS

```
call "xsm_bitop" using field-number, action, bit giving status.
call "xsm_cl_all_mdts".
```

uinstall

install an application function

SYNOPSIS

```
copy "sminstfn.incl.cobol".

77 usage              pic S(9)9 comp-5.
77 func-name          display-2 pic x(256).
77 func               entry.
77 language           pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_uinstall" using  usage, func-name, func, language
       giving status.
```

DESCRIPTION

This function installs an application routine that will be called from JAM library functions. Installation enables JAM to pass control to your code in the proper function context.

The possible values for `usage` are defined in the table below (and in the file: `smfuncs.incl.cobol`). See section 2.1.1. for more detailed descriptions of the various function types.

If an application is bound with the *-retain_all* option, then JAM can find the entrypoint `func` from the name. Most functions will install themselves automatically the first time they are called. Functions may also be explicitly installed. `func-name` is the name of the function. Use the operating system subroutine `s$find_entry` to find the entry point, or use the variant `xsm_n_uinstall`, which will find it for you. `language` should be set to 1 when programming in COBOL.

Value for `usage`	*Function type*	*Section – Page*
`UINIT-FUNC`	Initialization	2.2.10. – p. 22
`URESET-FUNC`	Reset	2.2.10. – p. 22
`VPROC-FUNC`	Video processing	2.2.13. – p. 25
`CKDIGIT-FUNC`	Check digit computation	2.2.9. – p. 22
`KEYCHG-FUNC`	Keychange	2.2.5. – p. 18

Value for usage	*Function type*	*Section – Page*
INSCRSR-FUNC	Insert/overwrite toggle	2.2.2. – p. 11
PLAY-FUNC	Playback recorded keys	2.2.11. – p. 23
RECORD-FUNC	Record keys for playback	2.2.11. – p. 23
AVAIL-FUNC	Check for recorded keys	2.2.11. – p. 23
BLKDRVR-FUNC	Block Driver function	
STAT-FUNC	Status line function	2.2.12. – p. 24
DFLT-FIELD-FUNC	Default Field function	2.2.2. – p. 11
DFLT-SCREEN-FUNC	Default Screen function	2.2.3. – p. 15
DFLT-SCROLL-FUNC	Default Scroll driver	
DFLT-GROUP-FUNC	Default Group function	2.2.6. – p. 19

RETURNS

1 if function was successfully installed.
-1 if malloc failure occurred.

VARIANTS

```
call "xsm_n_uinstall" using  usage, func-name, language giving
      status.
```

RELATED FUNCTIONS

```
call "xsm_async" using func, timeout.
```

ungetkey

push back a translated key on the input

SYNOPSIS

```
copy "smkeys.incl.cobol".

77 key                pic S(9)9 comp-5.
77 return-value       pic S(9)9 comp-5.
call "xsm_ungetkey" using key giving return-value.
```

DESCRIPTION

This function saves the translated key given by `key` so that it will be retrieved by the next call to `xsm_getkey`. Multiple calls are permitted. The key values are pushed onto a stack (LIFO).

When `xsm_getkey` reads a key *from the keyboard*, it flushes the display first, so that the operator sees a fully updated display before typing anything. Such is not the case for keys pushed back by `xsm_ungetkey`; since the input is coming from the program, it is responsible for updating the display itself.

RETURNS

The value of its argument, or
-1 if memory for the stack is unavailable.

RELATED FUNCTIONS

```
call "xsm_getkey" giving key.
```

unsvscreen

remove screens from the save list

SYNOPSIS

```
77 screen-list        display-2 pic x(256).
77 count              pic S(9)9 comp-5.
call "xsm_unsvscreen" using screen-list, count.
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. This function is used to remove screens from the save list. The argument `count` specifies the number of screens to be removed from the save list. See `xsm_svscreen`.

This function can be used at any point within your code. It is not necessary for the screen to be open at the time of the call. Any memory allocated to hold the screen is freed at the time of the call unless the screen is open. The memory associated with an open screen is de-allocated when that screen is closed. If a screen is not on the save list, a call to `xsm_unsvscreen` has no effect.

RELATED FUNCTIONS

```
call "xsm_issv" using screen-name giving status.
call "xsm_svscreen" using screen-list, count giving status.
```

viewport

modify viewport size and offset

SYNOPSIS

```
77 position-row       pic S(9)9 comp-5.
77 position-col       pic S(9)9 comp-5.
77 size-row           pic S(9)9 comp-5.
77 size-col           pic S(9)9 comp-5.
77 offset-row         pic S(9)9 comp-5.
77 offset-col         pic S(9)9 comp-5.
call "xsm_viewport" using position-row, position-col, size-row,
       size-col, offset-row, offset-col.
```

DESCRIPTION

This function dynamically sizes the current screen's viewport. A viewport has a maximum size of the screen or physical display – whichever is smaller. Use `size-row` and `size-column` to specify the number of rows and columns, respectively.

You can position the viewport anywhere on the physical display. To do this, think of your physical display as a grid made up of rows and columns that are one character apart. The top left corner of your screen monitor is at position row 0, column 0 . Now use the arguments `position-row` and `position-col` to specify the coordinates of the viewport's position.

Likewise, you can also specify which row and column of the screen will initially appear at top left corner of the viewport. Again starting at row 0, column 0, count from the top left of the screen to get the coordinates for `offset-row` and `offset-col`.

This function performs range checks on all parameters and suitably modifies them if necessary. In particular, be aware that a non–positive value of `size-row` and `size-col` will set the viewport to the maximum size in that dimension.

vinit

initialize video translation tables

SYNOPSIS

```
copy "vidfile.incl.cobol".

77 status              pic S(9)9 comp-5.
call "xsm_vinit" using VIDEO-ADDRESS giving status.
```

DESCRIPTION

This routine is called by `xsm_initcrt` as part of the initialization process. It can also be called directly by an application program. `VIDEO-ADDRESS` is the address of a key translation table contained in `vidfile.incl.cobol`, created using the `key2bin` and `bin2cob` utilities.

RETURNS

0 if initialization is successful.
program exit if video file is invalid or if `VIDEO-ADDRESS` is zero and `SMVIDEO` is undefined.

Note: The variant `xsm_n_vinit` has no return value.

VARIANTS

```
call "xsm_n_vinit" using video-file.
```

wcount

obtain number of currently open windows

SYNOPSIS

```
77 return-value       pic S(9)9 comp-5
call "xsm_wcount" giving return-value.
```

DESCRIPTION

This function returns the number of windows currently open. The number is equivalent to the number of windows in the window stack.

To select the screen beneath the current window, subtract 1 from the value returned by `xsm_wcount`, and then use the result as the argument to `xsm_wselect`.

This routine is useful when you are bringing another window to the top of the window stack (making the window active) with `xsm_wselect`.

RETURNS

The number of windows.
0 if the base form is the only open screen.
–1 if there is no current screen.

RELATED FUNCTIONS

```
call "xsm_wselect" using window-number giving return-value.
```

wdeselect

restore the formerly active window

SYNOPSIS

```
77 status              pic S(9)9 comp-5.
call "xsm_wdeselect" giving status.
```

DESCRIPTION

This function restores a window to its original position in the window stack, after it has been moved to the top by a call to `xsm_wselect`. Information necessary to perform this task is saved during each call to `xsm_wselect`, but is not stacked. Therefore a call to this routine must follow a call to `xsm_wselect` if it is to properly restore the window to its original position. Note that `xsm_wdeselect` does not have to be called if the window ordering on the stack is acceptable.

RETURNS

-1 if there is no window to restore.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_sibling" using should-it-be.
call "xsm_wcount" giving return-value.
call "xsm_wselect" using window-number giving return-value.
```

window

display a window at a given position

SYNOPSIS

```
77 screen-name        display-2 pic x(256).
77 start-line         pic S(9)9 comp-5.
77 start-column       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_r_window" using screen-name, start-line, start-column
       giving status.


77 screen-name        display-2 pic x(256).
77 status             pic S(9)9 comp-5.
call "xsm_r_at_cur" using screen-name giving status.


copy "myscreen.incl.cobol".

77 start-line         pic S(9)9 comp-5.
77 start-column       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_d_window" using SCREEN-ADDRESS, start-line,
       start-column giving status.


copy "myscreen.incl.cobol".

77 status             pic S(9)9 comp-5.
call "xsm_d_at_cur" using SCREEN-ADDRESS giving status.


77 lib-desc           pic S(9)9 comp-5.
77 screen-name        display-2 pic x(256).
77 start-line         pic S(9)9 comp-5.
77 start-column       pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_l_window" using lib-desc, screen-name, start-line,
       start-column giving status.


77 lib-desc           pic S(9)9 comp-5.
77 screen-name        display-2 pic x(256).
```

```
77 status              pic S(9)9 comp-5.
call "xsm_l_at_cur" using lib-desc, screen-name giving status.
```

DESCRIPTION

This set of functions is primarily intended to be used by developers who are writing their own executive. To open a window while under the control of the **JAM** Executive, use a **JAM** control string or `xsm_jwindow`.

Use `xsm_d_window`, `xsm_l_window`, or `xsm_r_window` to display `screen-name` with its upper left–hand corner at the specified line and column. The line and column are counted *from zero*. If `start-line` is 1, the window is displayed starting at the *second* line of the screen.

Use `xsm_d_at_cur`, `xsm_l_at_cur`, and `xsm_r_at_cur` to display a window at the current cursor position, offset by one line to avoid hiding that line's current display.

Whatever part of the display the new window does not occupy will remain visible. However, only the topmost (active) window and its fields are accessible to keyboard entry and library routines. **JAM** will not allow the cursor outside the topmost window. If you wish to shuffle windows use `xsm_wselect`.

If the window will not fit on the display at the location you request, **JAM** will adjust its starting position. If the window would hang below the screen and you have placed its upper left–hand corner in the *top* half of the display, the window is simply moved up. If your starting position is in the *bottom* half of the screen, the lower left hand corner of the window is placed there. Similar adjustments are made in the horizontal direction.

When you use `xsm_r_window` the named screen is sought first in the memory–resident screen list, and if found there is displayed using `xsm_d_window`. It is next sought in all the open libraries, and if found is displayed using `xsm_l_window`. Next it is sought on disk in the current directory; then under the path supplied to `xsm_initcrt`; then in all the paths in the setup variable `SMPATH`. If any path exceeds 80 characters, it is skipped. If the entire search fails, this function displays an error message and returns.

You may save processing time by using `xsm_d_window` and `xsm_d_at_cur` to display screens that are memory–resident. Use `bin2c` to convert screens from disk files, which you can modify using `jxform`, to program data structures you can compile into your application. A memory–resident screen is never altered at run–time, and may therefore be made shareable on systems that provide for sharing read–only data. `xsm_r_window` and `xsm_r_at_cur` can also display memory–resident screens, if they are properly installed using `xsm_formlist`. Memory–resident screens are particularly useful in applications that have a limited number of screens, or in environments that have a slow disk (*e.g.* MS–DOS). `SCREEN-ADDRESS` is the address of the screen in memory.

You may also save processing time by using `xsm_l_window` and `xsm_l_at_cur` to display screens that are in a library. A library is a single file containing many screens

(and/or JPL modules and keysets). You can assemble one from individual screen files using the utility `formlib`. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on the number of paths searched.

The library descriptor, `lib-desc`, is an integer returned by `xsm_l_open`, which you must call before trying to read any screens from a library. Note that `xsm_r_window` and `xsm_r_at_cur` also search any open libraries.

If you want to display a form use `xsm_r_form` or one of its variants. Use `xsm_close_window` to close the window.

RETURNS

0 if no error occurred during display of the screen;
–1 if the screen file's format is incorrect;
–2 if the screen cannot be found;
–3 if the system ran out of memory but the previous screen was restored;
–5 is returned if, after the screen was cleared, the system ran out of memory.
–6 is returned if the library is corrupted.

RELATED FUNCTIONS

```
call "xsm_close_window" giving status.
call "xsm_r_form" using screen-name giving status.
call "xsm_jwindow" using screen-name giving status.
```

winsize

allow end–user to interactively move and resize a window

SYNOPSIS

```
77 status              pic S(9)9 comp-5.
call "xsm_winsize" giving status.
```

DESCRIPTION

Calling `xsm_winsize` has the same effect as if the end–user had just hit the VWPT (viewport) logical key. The viewport status line appears and the user can move, resize and change the offset of the screen as well as move to any sibling windows. When the end–user hits XMIT (transmit) the previous status line is restored. If you wish to resize the viewport yourself, use `xsm_viewport`.

In order for the end–user to able to move from one window to another, the windows must be siblings. Windows are defined as siblings of one another either with `xsm_sibling` or by calling up a window as a sibling with a JAM control string. See the sections on "Viewports and Positioning" and "Control Strings" in the Author's Guide for further information.

RETURNS

–1 if call fails.
0 otherwise.

RELATED FUNCTIONS

```
call "xsm_sibling" using should-it-be.
call "xsm_viewport" using position-row, position-col, size-row,
       size-col, offset-row, offset-col.
```

wrecord

write data from the screen and LDB to a record defined in the data dictionary

SYNOPSIS

```
copy "record.incl.cobol".

77 record-name        display-2 pic x(256).
77 byte-count         pic S(9)9 comp-5.
call "xsm_wrecord" using RECORD, record-name, byte-count.
```

DESCRIPTION

This function writes data from fields within the current screen to a record that was defined in the data dictionary. If a field is not on the current screen, then the data is read from the LDB. This routine is commonly used with `xsm_rrecord`, which reads data from a record that was defined in the data dictionary. If you wish to write data only from the current screen, use `xsm_wrtstruct`. To write data from a group of consecutively numbered fields, use `xsm_wrt_part`. Use `xsm_getfield` to write information from an individual field to a string.

The record `RECORD`, contained in the file `record.incl.cobol` can be created from the data dictionary file `data.dic` via the `dd2struct` utility as follows:

```
dd2struct -gCOBOL data.dic
```

Each record item is a field of the type specified in the Data Dictionary Editor. See "Data Type" in the Author's Guide and `dd2struct` in the Utilities Guide for further information.

Once created, the record declarations may be treated exactly like any other record declarations. The argument `record-name` is the name of the data dictionary entry from which the record was created.

The argument `byte-count` is an integer. Upon return from `xsm_wrecord`, the value contained in the integer will be the number of bytes or characters written to the record. It will be 0 if an error occurred.

RELATED FUNCTIONS

```
call "xsm_putfield" using field-number, data giving status.
call "xsm_rrecord" using RECORD, record-name, byte-count.
```

wrt_part

write part of the screen to a record

SYNOPSIS

```
copy "screen.jam.incl.cobol".

77 first-field        pic S(9)9 comp-5.
77 last-field         pic S(9)9 comp-5.
call "xsm_wrt_part" using SCREEN, first-field, last-field.
```

DESCRIPTION

This function writes the contents of all fields between `first-field` and `last-field` to a record. An array and its scrolling occurrences will be copied only if the *first* element falls between `first-field` and `last-field`. Group selections are not copied. This routine is commonly used with `xsm_rd_part`, which reads part of a record into the current screen. If you wish to write the contents of all of the fields within the screen use `xsm_wrtstruct`. To write information to a record defined in the data dictionary, use `xsm_wrecord`. Use `xsm_getfield` to write information from an individual field to a string.

The record SCREEN, contained in the file `screen.jam.incl.cobol` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gCOBOL screen.jam
```

Each item in the record is a field of the type specified in the Screen Editor. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other record declaration. You can ignore the items that represent fields that do not fall within the bounds of the specified fields. However, the record definition must contain all of the fields on screen.

The arguments that represent the range of fields to be copied, `first-field` and `last-field` are passed as field numbers.

Remember, you must update the record declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
call "xsm_putfield" using field-number, data giving status.
```

```
call "xsm_rd_part" using SCREEN, first-field, last-field
call "xsm_wrtstruct" using SCREEN, byte-count.
```

wrtstruct

write data from the screen to a record

SYNOPSIS

```
copy "screen.jam.incl.cobol".

77 byte-count          pic S(9)9 comp-5.
call "xsm_wrtstruct" using SCREEN, byte-count.
```

DESCRIPTION

This function writes the contents of all of the fields within the current screen to a record. It will not copy group selections. This routine is commonly used with `xsm_rdstruct` which reads data from a record to all of the fields within the current screen. If you wish to write the contents of a group of consecutively numbered fields into a record use `xsm_wrt_part`. To write information to a record defined in the data dictionary, use `xsm_wrecord`. Use `xsm_getfield` to write the contents of an individual field into a string.

The record SCREEN, contained in the file `screen.jam.incl.cobol` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gCOBOL screen.jam
```

Each item in the record is a field of the type specified in the Screen Editor. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other record declaration. If you specify the type `omit`, data will not be written into the field.

The argument `byte-count` is the address of an integer variable. `xsm_wrtstruct` will store there the number of bytes copied to the record.

Remember, you must update the record declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
call "xsm_putfield" using field-number, data giving status.
call "xsm_rd_struct" using screen, byte-count.
call "xsm_wrt_part" using SCREEN, first-field, last-field.
```

wselect

activate a window

SYNOPSIS

```
77 window-number      pic S(9)9 comp-5.
77 return-value       pic S(9)9 comp-5.
call "xsm_wselect" using window-number giving return-value.
```

DESCRIPTION

Although **JAM** allows you to display multiple windows at one time, only one window may be active. Windows may overlap each other, or may be *tiled* (no overlap). The window at the top of the window stack is the active window, and the only window accessible to library routines and keyboard entry. Use `xsm_wselect` to bring a window to the active position on top of the window stack. If any of the referenced window is hidden by an overlying window, it will be brought to the forefront of the display. In either case, the cursor is placed within the window. **JAM** will restore the cursor to its position when the screen was most recently de-activated.

The window to be activated is referenced by its number in the window stack. Windows are numbered sequentially, starting from the bottom of the stack. The form underlying all the windows (the base form) is window `0`, the first window displayed is `1` and so forth. Since a screen's number depends on its position on the window stack, calling `xsm_wselect` will alter a window's number as well as it position on the stack.

Alternatively, windows may be referenced by their screen name with the variant `xsm_n_wselect`. If you use this routine, you do not have to worry about keeping track of the non-active window's position on the stack. However, `xsm_n_wselect` will not find windows displayed with `xsm_d_window` or related functions, because they do not record the screen name.

Here are two different ways of using window selection. One way to use this is to select a hidden screen, update it (using `xsm_putfield`) and deselect it (using `xsm_wdeselect`). The portion of the hidden screen that is visible will be updated with the new data. Because of *delayed write* the update will be done when the next keyboard input is sought. The other method is to select a hidden screen and open the keyboard; in this case, the selected screen becomes visible, and may hide part or all of the screen that was previously active. In this way you can implement multi-page forms, or switch among several windows that *tile* the screen (do not overlap).

RETURNS

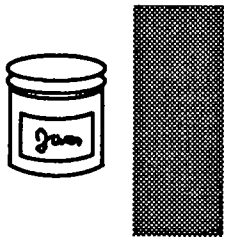
The number of the window that was made active (either the number passed, or the maximum if that was out of range).
–1 if the window was not found or the window was not open.

VARIANTS

```
call "xsm_n_wselect" using window-name giving return-value.
```

RELATED FUNCTIONS

```
call "xsm_sibling" using should-it-be.
call "xsm_wcount" giving return-value.
call "xsm_wdeselect" giving status.
```



Chapter 13.
Library Function Index

This chapter lists all JAM library functions, sorted by name. Function names appear on the left, and the section of the Function Reference Chapter in which the function is described appears on the right.

"xsm_e_ltofield" using field-name, element, value giving status. **ltofield**
"xsm_e_novalbit" using field-name, element giving status. ..... **novalbit**
"xsm_e_null" using field-name, element giving status. ..... **null**
"xsm_e_off_gofield" using field-name, element, offset
giving status. ..... **off_gofield**
"xsm_e_protect" using field-name, element giving status. ..... **protect**
"xsm_e_putfield" using name, element, data giving status. ..... **putfield**
"xsm_e_unprotect" using field-name, element giving status. ..... **protect**
"xsm_edit_ptr" using field-number, edit-type giving buffer. .... **edit_ptr**
"xsm_emsg" using message. ..... **emsg**
"xsm_err_reset" using message. ..... **err_reset**
"xsm_fi_path" using file-name giving buffer. ..... **fi_path**
"xsm_finquire" using field-number, which giving value. ..... **finquire**
"xsm_flush". ..... **flush**
"xsm_formlist" using name, address giving status. ..... **formlist**
"xsm_ftog" using field-number, group-occurrence giving buffer. ..... **ftog**
"xsm_ftype" using field-number, precision-ptr giving type. ..... **ftype**
"xsm_fval" using field-number giving status. ..... **fval**
"xsm_getcurno" giving field-number. ..... **getcurno**
"xsm_getfield" using buffer, field-number giving length. ..... **getfield**
"xsm_getjctrl" using key, default giving buffer. ..... **getjctrl**
"xsm_getkey" giving key. ..... **getkey**
"xsm_gofield" using field-number giving status. ..... **gofield**
"xsm_gp_inquire" using group-name, which giving value. ..... **gp_inquire**
"xsm_gwrap" using buffer, field-number, buffer-length giving length.**gwrap**
"xsm_hlp_by_name" using help-screen giving status. ..... **hlp_by_name**
"xsm_home" giving field-number. ..... **home**
"xsm_i_..." using field-name, occurrence, ... **i_**
"xsm_i_achg" using field-name, occurrence, display-attribute
giving status. ..... **achg**
"xsm_i_amt_format" using field-name, occurrence, buffer
giving status. ..... **amt_format**
"xsm_i_bitop" using array-name, occurrence, action, bit
giving status. ..... **bitop**
"xsm_i_dblval" using field-name, occurrence giving value. ..... **dblval**
"xsm_i_dlength" using field-name, occurrence giving data-length. **dlength**
"xsm_i_doccur" using field-name, occurrence, count
giving return-value. ..... **doccur**
"xsm_i_dtofield" using field-name, occurrence, value, format
giving status. ..... **dtofield**
"xsm_i_finquire" using field-name, occurrence, which
giving value. ..... **finquire**
"xsm_i_fldno" using field-name, occurrence giving field-number. ... **fldno**
"xsm_i_ftog" using field-name, occurrence, group-occurrence
giving buffer. ..... **ftoq**
"xsm_i_fval" using field-name, occurrence giving status. ..... **fval**

```
"xsm_i_getfield" using buffer, name, occurrence giving length. . getfield
"xsm_i_gofield" using field-name, occurrence giving status. ..... gofield
"xsm_i_gtof" using group-name, group-occurrence, occurrence
          giving field-number. ..................................... gtof
"xsm_i_intval" using field-name, occurrence giving value. ........ intval
"xsm_i_ioccur" using field-name, occurrence, count
          giving lines-inserted. ................................... ioccur
"xsm_i_is_no" using field-name, occurrence giving status. ......... is_no
"xsm_i_is_yes" using field-name, occurrence giving status. ....... is_yes
"xsm_i_itofield" using field-name, occurrence, value
          giving status. ........................................... itofield
"xsm_i_lngval" using field-name, occurrence giving value. ........ lngval
"xsm_i_ltofield" using field-name, occurrence, value
          giving status. ........................................... ltofield
"xsm_i_novalbit" using field-name, occurrence giving status. ... novalbit
"xsm_i_null" using field-name, occurrence giving status. ........... null
"xsm_i_off_gofield" using field-name, occurrence, offset
          giving status. ........................................... off_gofield
"xsm_i_putfield" using name, occurrence, data giving status. ... putfield
"xsm_ininames" using name-list giving status. ..................... ininames
"xsm_initcrt" using path. ......................................... initcrt
"xsm_input" using initial-mode giving key. ........................ input
"xsm_inquire" using which giving value. ........................... inquire
"xsm_intval" using field-number giving value. ..................... intval
"xsm_is_no" using field-number giving status. ...................... is_no
"xsm_is_yes" using field-number giving status. ..................... is_yes
"xsm_isabort" using flag giving OLD-FLAG. ......................... isabort
"xsm_iset" using which, newval giving value. ....................... iset
"xsm_isselected" using group-name, group-occurrence
          giving status. ........................................... isselected
"xsm_issv" using screen-name giving status. ........................ issv
"xsm_itofield" using field-number, value giving status. ........ itofield
"xsm_jclose" giving status. ....................................... jclose
"xsm_jform" using screen-name giving status. ....................... jform
"xsm_jinitcrt" using path. ........................................ initcrt
"xsm_jplcall" using jplcall-text giving return-value. ........... jplcall
"xsm_jplload" using module-name-list giving status. ............. jplload
"xsm_jplpublic" using module-name-list giving status. ......... jplpublic
"xsm_jplunload" using module-name giving status. ............... jplunload
"xsm_jresetcrt". ..................................................... resetcrt
"xsm_jtop" using screen-name giving status. ........................ jtop
"xsm_jwindow" using screen-name giving status. ................... jwindow
"xsm_jxinitcrt" using path. ....................................... initcrt
"xsm_jxresetcrt". ..................................................... resetcrt
"xsm_keyfilter" using flag giving old-flag. ..................... keyfilter
```

```
"xsm_o_bitop" using field-number, occurrence, action, bit
            giving status. ..................................... bitop
"xsm_o_chg_attr" using field-number, element, display-attribute
            giving status. ..................................... chg_attr
"xsm_o_dblval" using field-number, occurrence giving value. ..... dblval
"xsm_o_dlength" using field-number, occurrence giving data-length.dlength
"xsm_o_doccur" using field-number, occurrence, count
            giving return-value. ............................... doccur
"xsm_o_dtofield" using field-number, occurrence, value, format
            giving status. ..................................... dtofield
"xsm_o_finquire" using field-number, occurrence, which
            giving value. ...................................... finquire
"xsm_o_fldno" using field-number, occurrence giving field-number. . fldno
"xsm_o_ftog" using field-number, occurrence, group-occurrence
            giving buffer. ..................................... ftog
"xsm_o_fval" using field-number, occurrence giving status. ..... fval
"xsm_o_getfield" using buffer, field-number, occurrence
            giving length. ..................................... getfield
"xsm_o_gofield" using field-number, occurrence giving status. ... gofield
"xsm_o_gwrap" using buffer, field-number, occurrence, buffer-length
            giving status. ..................................... gwrap
"xsm_o_intval" using field-number, occurrence giving value. ..... intval
"xsm_o_ioccur" using field-number, occurrence, count
            giving lines-inserted. ............................. ioccur
"xsm_o_is_no" using field-number, occurrence giving status. ..... is_no
"xsm_o_is_yes" using field-number, occurrence giving status. ..... is_yes
"xsm_o_itofield" using field-number, occurrence, value
            giving status. ..................................... itofield
"xsm_o_lngval" using field-number, occurrence giving value. ..... lngval
"xsm_o_ltofield" using field-number, occurrence, value
            giving status. ..................................... ltofield
"xsm_o_novalbit" using field-number, occurrence giving status. . novalbit
"xsm_o_null" using field-number, occurrence giving status. ..... null
"xsm_o_off_gofield" using field-number, occurrence, offset
            giving status. ..................................... off_gofield
"xsm_o_putfield" using field-number, occurrence, data
            giving status. ..................................... putfield
"xsm_o_pwrap" using field-number, occurrence, text giving status. . pwrap
"xsm_occur_no" giving occurrence. ............................... occurno
"xsm_off_gofield" using field-number, offset giving status. . off_gofield
"xsm_option" using option, newval giving oldval. ................ option
"xsm_oshift" using field-number, offset giving return-value. ..... oshift
"xsm_pinquire" using which giving buffer. ....................... pinquire
"xsm_protect" using field-number giving status. ................. protect
"xsm_pset" using which, newval giving buffer. ..................... pset
"xsm_putfield" using field-number, data giving status. ......... putfield
"xsm_putjctrl" using key, control-string, default giving status. putjctrl
```

```
"xsm_tst_all_mdts" using occurrence giving field-number. ... tst_all_mdts
"xsm_uinstall" using  usage, func, func-name giving status. .... uinstall
"xsm_ungetkey" using key giving return-value. ..................... ungetkey
"xsm_unprotect" using field-number giving status. ................ protect
" xsm_unsvscreen" using screen-list, count. ..................... unsvscreen
"xsm_viewport" using position-row, position-col, size-row,
          size-col, offset-row, offset-col. ..................... viewport
"xsm_vinit" using VIDEO-ADDRESS giving status. ..................... vinit
"xsm_wcount" giving return-value. ................................ wcount
"xsm_wdeselect" giving status. ................................ wdeselect
"xsm_winsize" giving status. ..................................... winsize
"xsm_wrecord" using RECORD, record-name, byte-count. ............. wrecord
"xsm_wrt_part" using SCREEN, first-field, last-field. ........... wrt_part
"xsm_wrtstruct" using SCREEN, byte-count. ..................... wrtstruct
"xsm_wselect" using window-number giving return-value. ........... wselect
```

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

T

U

V

W

Appendix A.
Notes for C Programmers

The following notes highlight the differences between this COBOL language guide and the standard JAM library, written in C. It is intended as a guide for programmers who are already familiar with the C version of JAM.

A.1.
INTRODUCTION

The COBOL interface library is written in C. It consists primarily of interface functions which can be called directly from COBOL programs. The interface functions first set up the appropriate arguments for the corresponding functions in the standard JAM library. They then call the standard library function, and finally set up suitable return values for COBOL. Each interface function is named for the standard library function that it calls, with the `sm_` prefix changed to `xsm_`. Thus, `xsm_getfield` is the interface function to the standard library routine `sm_getfield`.

A.2.
SYNTAX

A.2.1.
Numeric Arguments

All COBOL arguments are passed by reference. That is, the COBOL interface functions expect `char *`'s and `int *`'s (rather than `int`'s), and pass the integer values to their respective standard library functions.

A.2.2.
Character String Arguments

For standard library functions that are passed character strings, the interface function makes a null–terminated copy of the data in a static buffer, which it then uses to call the standard library function.

A.2.3.
Return Values From Library Routines

Under the Stratus COBOL compiler, an integer or string value can be returned directly to a COBOL function from the C function it calls by including a `giving` phrase in the `call` statement.

A.3. UNSUPPORTED STANDARD LIBRARY FUNCTIONS

The following functions are not supported in the Stratus COBOL interface.

`do_uinstalls` See `install`.

`fi_open` This function is not supported in the COBOL interface.

`formlist` The memory–resident screen list is not supported in the COBOL interface.

`fptr` Supported, but not documented. Since the COBOL calling function must supply its own buffer, this function is essentially equivalent to `getfield`. Use `getfield` instead.

`install` The COBOL interface has three methods for installing functions, depending on their type (see section 2.1.1. for further information):

- Asynchronous functions are installed with the special interface function `xsm_async`. See the Function Reference listing for details.
- Field, screen, group, and control functions are installed via the `-retain_all` argument to the `bind` command.
- All other types of functions (including the *default* field, screen, and group functions) are installed with the special interface function

`xsm_n_uinstall`. These are installed by type and function name, rather than through a pointer to a structure. See the Function Reference listing for details.

`kslabel` This function is not supported in the COBOL interface.

`lngval` This function is unnecessary in the COBOL interface, since longs and ints are defined to have the same length. Use `intval` instead.

`ltofield` Use `itofield` instead. See explanation for `lngval` above.

`rs_data` Unsupported in COBOL.

`save_data` Unsupported in COBOL.

`sv_data` Unsupported in COBOL

`sv_free` Unsupported in the COBOL interface.

A.4. SPECIAL INTERFACE LIBRARY FUNCTIONS

`async` Installs an asynchronous function. See page 95 for details.

`uinstall` Install hook functions that are not attached to screens. See page 296 for details.

A.5. FUNCTIONAL DIFFERENCES IN SUPPORTED LIBRARY FUNCTIONS

`d_at_cur` See `d_form`.

`d_form` The program data structure created by `bin2cob` must be included in the call to the interface function with the statement:

```
copy "myscreen.incl.cobol".
```

`d_keyset` See `d_form`.

`msg_get` Message mnemonics are not supported. Messages are divided into classes based on their numbers, with up to 4096 messages per class. The message

class is the message number divided by 4096, and the message offset within the class is the remainder. Predefined JAM message numbers and classes are shown in Appendix B..

`rd_part` See `rdstruct`.

`rdstruct` The interface function receives a structure pointer. It is the programmer's responsibility to provide the proper structure. COBOL structures can be generated with the utility `f2struct` as follows:

```
f2struct -gCOBOL screenname
```

The structure can be inserted into a COBOL program via the `copy` statement. No language argument is passed to the interface function. The language is understood to be COBOL.

`rrecord` Like the interface function for `rdstruct`, this interface function receives a buffer pointer. A structure can be generated for this function with the utility `dd2struct`, by specifying the COBOL language option., and then inserted into a COBOL program via the `copy` command No language argument is passed to the interface function The language is understood to be COBOL.

`vinit` See `keyinit`.

`d_window` See `d_form`.

`wrt_part` See `rdstruct`.

`wrecord` See `rrecord`.

`wrtstruct` See `rdstruct`.

A.6. RETURN VALUES FROM COBOL FUNCTIONS

JAM expects every user-installed function to return an integer, even if the return value is ignored. In the Stratus version of the COBOL language interface, COBOL procedures move the desired return value to a variable, say for example, `ret-value`. The function is then exited with the following statement:

```
exit program with ret-value.
```

A.7. HEADER FILES

The COBOL interface header files contain actual data rathar than mnemonics. Note that all underscore characters that normally appear in C language header files appear as dashes in COBOL versions. Some header files may be different than their C counterparts. The following header files are available:

File	*Contents*
smattrib.incl.cobol	Display attribute values.
smbitops.incl.cobol	Values for use with the library routine `xsm_bitop`.
smedits.incl.cobol	Values for field edits.
smerror.incl.cobol	Values for use with the message functions.
smflags.incl.cobol	Context bit values for entry, exit and validation hook functions.
smfuncs.incl.cobol	Values and flags for function installation.
smglobal.incl.cobol	Values for obtaining global data (unused in release 5).
smglobs.incl.cobol	Values for variable inquiry functions.
smkeys.incl.cobol	Values for JAM key codes. Needed to handle returns from key board processing routines.
smmisc.incl.cobol	Miscellaneous values.
smmisc2.incl.cobol	Miscellaneous values.
smmisc3.incl.cobol	Miscellaneous values.
smsoftk.incl.cobol	Values for use with keyset routines.
smumisc.incl.cobol	Miscellaneous values.
smvideo.incl.cobol	Values for video file entries.

Appendix B.
Error Message Numbers

The following tables illustrate the **JAM** message numbers and classes for use with the message functions `xsm_msgfind` and `xsm_msg_get`. The first column shows the message mnemonic, which is not used in the COBOL interface, but which is included for convenience. The second contains the message number, showing how it is computed, by multiplying the message class by 4096 and then adding the message offset. The third column contains the text of the message.

Refer to the *Configuration Guide* for information on creating and editing message files.

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
Initialization Messages		
SM-ENTERTERM	8*4096 + 0	Please enter terminal type or <RETURN> to exit.
SM-MALLOC	8*4096 + 1	Insufficient memory available
SM-CANCEL	8*4096 + 2	Terminated.
SM-BADTERM	8*4096 + 6	Unknown terminal type
Math Messages		
SM-FNUM	8*4096 + 8	Bad field # or subscript.
SM-DZERO	8*4096 + 9	Divide by zero.
SM-EXPONENT	8*4096 + 10	Exponentiation invalid.

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
SM-INVDATE	8*4096 + 11	Invalid date.
SM-MATHERR	8*4096 + 12	Math or JPL error
SM-FORMAT	8*4096 + 116	Invalid format.
SM-DESTINATION	8*4096 + 117	Invalid destination.
SM-INCOMPLETE	8*4096 + 118	Expression incomplete.
SM-ORAND	8*4096 + 119	Operand expected.
SM-ORATOR	8*4096 + 120	Operator expected.
SM-EXTRAPARENS	8*4096 + 121	Right parenthesis unexpected
SM-MISSPARENS	8*4096 + 122	Right parenthesis expected.
SM-DEEP	8*4096 + 123	Formula too complicated.
SM-FUNCTION	8*4096 + 124	Invalid function.
SM-ARGUMENT	8*4096 + 125	Invalid argument.
SM-MISMATCH	8*4096 + 126	Type mismatch.
SM-NOTMATH	8*4096 + 127	Not a math expression.
SM-QUOTE	8*4096 + 128	Missing quote character.
SM-SYNTAX	8*4096 + 129	Syntax error.
Read Window Messages		
SM-FRMDATA	8*4096 + 13	Bad data in screen.
SM-NOFORM	8*4096 + 14	Cannot find screen.
SM-FRMERR	8*4096 + 15	Error while reading screen.
SM-BIGFORM	8*4096 + 16	Screen has fields that extend beyond display size.

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
err_reset Messages		
SM-ERROR	8*4096 + 19	ERROR:
SM-SP1	8*4096 + 20	Please hit the space bar
SM-SP2	8*4096 + 21	after reading this message
Field Validation Messages		
SM-RENTRY	8*4096 + 22	Entry is required.
SM-MUSTFILL	8*4096 + 23	Must fill field.
SM-AFOVRFLW	8*4096 + 24	Amount field overflow.
Group Validation Messages		
SM-ONLYONE	8*4096 + 76	Select exactly one item.
ckdigit Messages		
SM-TOO-FEW-DIGITS	8*4096 + 25	Too few digits.
SM-CKDIGIT	8*4096 + 26	Check digit error.
Help Messages		
SM-HITANY	8*4096 + 27	Hit any key to continue.
SM-NOHELP	8*4096 + 29	No help text available.
SM-MAXHELP	8*4096 + 30	Five help levels maximum.
SM-FRMHELP	8*4096 + 73	No screen-level help text available
Range Messages		
SM-OUTRANGE	8*4096 + 31	Out of range.
Date/Time Messages		
SM-SYSDATE	8*4096 + 39	Use clear for system date/time.

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
SM-DATFRM	8*4096 + 40	Invalid date/time format.
SM-DATCLR	8*4096 + 41	Invalid date/time; clear gets system date.
SM-DATINV	8*4096 + 42	Invalid date/time; enter a valid date/time.
SM-KSDATA	8*4096 + 43	Bad data in keyset.
SM-KSERR	8*4096 + 44	Error while reading keyset.
SM-KSNONE	8*4096 + 45	Cannot find keyset.
SM-KSMORE	8*4096 + 46	MORE logo for key label
Day Of Week Abbreviations (1–7 = Sun–Sat)		
SM-DAYA1	8*4096 + 47	
SM-DAYA2	8*4096 + 48	
SM-DAYA3	8*4096 + 49	
SM-DAYA4	8*4096 + 50	
SM-DAYA5	8*4096 + 51	
SM-DAYA6	8*4096 + 52	
SM-DAYA7	8*4096 + 53	
Day of Week Spelled Out (1–7 = Sun–Sat)		
SM-DAYL1	8*4096 + 54	
SM-DAYL2	8*4096 + 55	
SM-DAYL3	8*4096 + 56	
SM-DAYL4	8*4096 + 57	
SM-DAYL5	8*4096 + 58	

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
SM-DAYL6	8*4096 + 59	
SM-DAYL7	8*4096 + 60	
Scroll Messages		
SM-MOREDATA	8*4096 + 66	No more data.
SM-SCRLMEM	8*4096 + 67	Insufficient memory for scrolling.
input **Messages**		
SM-READY	8*4096 + 68	Ready
SM-WAIT	8*4096 + 69	Wait
SM-YES	8*4096 + 70	y
SM-NO	8*4096 + 71	n
SM-SEARCH	8*4096 + 74	Searching for: '%s'
Local Print Message		
SM-NOTEMP	8*4096 + 72	Cannot open temporary file.
Window Resizing (Must Be In Order)		
SM-WMSMOVE	8*4096 + 77	MOVE resize offset
SM-WMSSIZE	8*4096 + 78	move RESIZE offset
SM-WMSOFF	8*4096 + 79	move resize OFFSET
SM-LPRINT	8*4096 + 80	Local print file %s created.
SM-NOFILE	8*4096 + 82	Could not open file
Regular Expression Validation (Character Mask)		
SM-RX1	8*4096 + 86	Invalid character.
SM-RX2	8*4096 + 87	Incomplete entry.

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
SM-RX3	8*4096 + 88	No more input allowed.
SM-TABLOOK	8*4096 + 90	Invalid entry.
JPL Messages		
SM-ILLELSE	8*4096 + 98	Illegal Else
SM-NUMBER	8*4096 + 99	Illegal Number
SM-EOT	8*4096 + 100	unexpected End Of File
SM-BREAK	8*4096 + 101	BREAK not within loop
SM-NOARGS	8*4096 + 102	Verb needs arguments
SM-BIGVAR	8*4096 + 103	Variable size larger than 255
SM-EXCESS	8*4096 + 104	Extra data at end of line
SM-EOL	8*4096 + 105	Source line too long
SM-FILEIO	8*4096 + 106	System File I/O error
SM-FOR	8*4096 + 107	USAGE: FOR varname = Value WHILE (expression) STEP [+–]value
SM-LINE-2-LONG	8*4096 + 108	Line too long after expansion
SM-RCURLY	8*4096 + 109	Ended block not begun
SM-NONAME	8*4096 + 110	Expected variable name
SM-NOTARGET	8*4096 + 111	Target does not exist
SM-1JPL-ERR	8*4096 + 112	%s at line %d in %s: '%s' First %s gets error message, such as SM–LINE–2–LONG, %d is line number, next %s gets SM–JSRCFILE or SM–JPLATCH, last %s gets a quote from the offending line.

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
SM-2JPL-ERR	8*4096 + 113	%s in %s First %s gets SM-JSRCFILE or SM-JPLATCH, second gets SM-MALLOC or other not easily traceable error.
SM-JPLATCH	8*4096 + 115	attached JPL procedure
NOTE: Message Numbers 116 – 129 Used For Math (see above)		
SM-NEXT	8*4096 + 130	next not in a loop
SM-VERB-UNKNOWN	8*4096 + 131	unknown verb
SM-JPLFORM	8*4096 + 132	screen JPL procedure
SM-NOT-LOADED	8*4096 + 133	file not loaded
Command Line		
SM-GA-FLG	8*4096 + 134	Illegal flag
SM-GA-CHAR	8*4096 + 135	Flag must be followed by a char
SM-GA-ARG	8*4096 + 136	Flag has no argument
SM-GA-DIG	8*4096 + 137	Flag must be followed by digits
Function Installation		
SM-NOFUNC	8*4096 + 138	Function not found.
SM-BADPROTO	8*4096 + 139	Bad prototype.
SM-JPLPUBLIC	8*4096 + 140	public JPL procedure
Missing Screen Library		
SM-NO-LIB	8*4096 + 141	Library not found
Default Null Edit		
SM-NULLEDIT	8*4096 + 142	NULL

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
SM-RP-NULL	8*4096 + 143	n
JAM/DB*i*		
SM-DBI-NOT-INST	8*4096 + 144	DBi not installed
**Msgs 145 – 170 Free For Use		
Month of Year Abbreviations (1–12)		
SM-MONA1	8*4096 + 171	
SM-MONA2	8*4096 + 172	
SM-MONA3	8*4096 + 173	
SM-MONA4	8*4096 + 174	
SM-MONA5	8*4096 + 175	
SM-MONA6	8*4096 + 176	
SM-MONA7	8*4096 + 177	
SM-MONA8	8*4096 + 178	
SM-MONA9	8*4096 + 179	
SM-MONA10	8*4096 + 180	
SM-MONA11	8*4096 + 181	
SM-MONA12	8*4096 + 182	
Month of Year Spelled Out (1–12)		
SM-MONL1	8*4096 + 183	
SM-MONL2	8*4096 + 184	
SM-MONL3	8*4096 + 185	
SM-MONL4	8*4096 + 186	

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
SM-MONL5	8*4096 + 187	
SM-MONL6	8*4096 + 188	
SM-MONL7	8*4096 + 189	
SM-MONL8	8*4096 + 190	
SM-MONL9	8*4096 + 191	
SM-MONL10	8*4096 + 192	
SM-MONL11	8*4096 + 193	
SM-MONL12	8*4096 + 194	
AM and PM representations		
SM-AM	8*4096 + 195	
SM-PM	8*4096 + 196	
Date/Time Formats		
SM-0DEF-DTIME	8*4096 + 197	%m/%d/%2y %h:%0M %p
SM-1DEF-DTIME	8*4096 + 198	%m/%d/%2y
SM-2DEF-DTIME	8*4096 + 199	%h:%0M %p
SM-3DEF-DTIME	8*4096 + 200	%m/%d/%2y %h:%0M %p
SM-4DEF-DTIME	8*4096 + 201	%m/%d/%2y %h:%0M %p
SM-5DEF-DTIME	8*4096 + 202	%m/%d/%2y %h:%0M %p
SM-6DEF-DTIME	8*4096 + 203	%m/%d/%2y %h:%0M %p
SM-7DEF-DTIME	8*4096 + 204	%m/%d/%2y %h:%0M %p
SM-8DEF-DTIME	8*4096 + 205	%m/%d/%2y %h:%0M %p
SM-9DEF-DTIME	8*4096 + 206	%m/%d/%2y %h:%0M %p

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
SM-CALC-DATE	8*4096 + 207	%m/%d/%4y
Block Mode Messages		
SM-BAD-DIGIT	8*4096 + 208	Bad character in digits only field.
SM-BAD-YN	8*4096 + 209	Bad character in yes/no field.
SM-BAD-ALPHA	8*4096 + 210	Bad character in letters only field.
SM-BAD-NUM	8*4096 + 211	Bad character in numeric field.
SM-BAD-ALPHNUM	8*4096 + 212	Bad character in alphanumeric field.
Currency Formats		
SM-DECIMAL	8*4096 + 213	.
SM-0DEF-CURR	8*4096 + 214	".22,1$
SM-1DEF-CURR	8*4096 + 215	".09,
SM-2DEF-CURR	8*4096 + 216	".09
SM-3DEF-CURR	8*4096 + 217	".09
SM-4DEF-CURR	8*4096 + 218	".09
SM-5DEF-CURR	8*4096 + 219	".09
SM-6DEF-CURR	8*4096 + 220	".09
SM-7DEF-CURR	8*4096 + 221	".09
SM-8DEF-CURR	8*4096 + 222	".09
SM-9DEF-CURR	8*4096 + 223	".09
Default Status Lines		
SM-1STATS	8*4096 + 224	default status line

Screen Manager Library Messages: SM-MSGS = 8		
Mnemonic	*Message Number*	*Message Text*
SM-12STATS	8*4096 + 225	ZOOM window status line
SM-VERNO	8*4096 + 226	5.01

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-2STATS	9*4096 + 3	Screen Editor MOVE status line
FM-3STATS	9*4096 + 4	Screen Editor COPY status line
FM-4STATS	9*4096 + 5	linedraw status line, pen down
FM-5STATS	9*4096 + 6	linedraw status line, pen up
FM-6STATS	9*4096 + 7	select mode status line
FM-7STATS	9*4096 + 8	group information status line
FM-8STATS	9*4096 + 9	group select mode status line
FM-9STATS	9*4096 + 75	group info status line for DD
FM-10STATS	9*4096 + 76	parallel array select status line
FM-11STATS	9*4096 + 119	JPL window status line
Sign–off Message		
FM-BYE	9*4096 + 10	
Characters For Vertical & Horizontal		
FM-VERT	9*4096 + 11	
FM-HORIZ	9*4096 + 12	
General Error Messages For Screen Editor		
FM-BADENTRY	9*4096 + 13	Bad entry.

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-MXSCRN	9*4096 + 14	Maximum number of %s ("lines" or "columns") on the screen is %d.
FM-MNBRDR	9*4096 + 15	Minimum number of %s to hold screen data and a border is %d.
FM-MNFORM	9*4096 + 16	Minimum number of %s to hold screen data is %d.
FM-NOMENU	9*4096 + 17	
FM-MINV	9*4096 + 18	Below minimum value of %d.
FM-THARRAY	9*4096 + 19	This array cannot fit on maximum size screen.
**Msgs 20–25 Free For Use		
Display Attributes		
FM-UTT0	9*4096 + 26	"NON-DISP "
FM-UTT1	9*4096 + 27	"REVERSE "
FM-UTT2	9*4096 + 28	"BLINKING "
FM-UTT3	9*4096 + 29	"UNDLN "
FM-UTT4	9*4096 + 30	"HILIGHT "
FM-UTT5	9*4096 + 31	"DIM "
FM-UTT6	9*4096 + 32	"STANDOUT "
FM-UTT7	9*4096 + 33	"ALTERNATE "
Colors		
FM-CLR0	9*4096 + 34	"BLACK "
FM-CLR1	9*4096 + 35	"BLUE "

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-CLR2	9*4096 + 36	"GREEN "
FM-CLR3	9*4096 + 37	"CYAN "
FM-CLR4	9*4096 + 38	"RED "
FM-CLR5	9*4096 + 39	"MAGENTA "
FM-CLR6	9*4096 + 40	"YELLOW "
FM-CLR7	9*4096 + 41	"WHITE "
**Msgs 42 – 45 Available For Use		
Opening and Saving Screens		
FM-TEMPLT	9*4096 + 46	Press <XMIT> to accept the template, <EXIT> to cancel.
FM-NOOPEN	9*4096 + 47	Cannot create screen %s.
FM-FSAVED	9*4096 + 48	Screen '%s' saved.
FM-WRFORM	9*4096 + 49	Error writing screen '%s'.
Exiting		
FM-QSAVE	9*4096 + 50	"Do you want to save the current screen? (enter y or n) "
FM-QCONT	9*4096 + 51	Do you want to continue processing? (enter y or n)
FM-QEXIT	9*4096 + 52	Do you really want to exit? (enter y or n)
Arrays		
FM-ARSET	9*4096 + 54	Cannot set array from this field.
FM-ARHROOM	9*4096 + 55	No room for horizontal array.

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-ARVROOM	9*4096 + 56	No room for vertical array.
FM-AROVERLAP	9*4096 + 58	Overlaps existing field.
Filters For Summary Window		
FM-FLT0	9*4096 + 59	unfilt
FM-FLT1	9*4096 + 60	digit
FM-FLT2	9*4096 + 61	yes/no
FM-FLT3	9*4096 + 62	letters
FM-FLT4	9*4096 + 63	numeric
FM-FLT5	9*4096 + 64	alphanum
FM-FLT6	9*4096 + 65	reg exp
**Msg 66 Reserved For FM-FLT7 If Ever Defined		
Miscellaneous Messages		
FM-RDO-BUTTON	9*4096 + 67	"RADIO-BUTTON "
FM-CHK-BOX	9*4096 + 68	"CHECK-BOX "
FM-PARALLEL	9*4096 + 71	"PARALLEL "
FM-UCSET	9*4096 + 72	Set upper or lower case
FM-MINMAX	9*4096 + 73	Minimum greater than maximum
FM-ONEYES	9*4096 + 74	Enter 'yes' for one option only
FM-PARNS	9*4096 + 77	Not a scrolling array
FM-PARDS	9*4096 + 78	Different size arrays
FM-PARDC	9*4096 + 79	Different CIRCULARity
FM-BK-POSTFIX	9*4096 + 80	"-BKGND"

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-PREV	9*4096 + 81	"PREV-FLD "
FM-SHRNG	9*4096 + 82	The shifting increment must be at least 1, but no more than
FM-FLDLEN	9*4096 + 83	Length must be non-zero and no greater than %d.
FM-WR-RJUST	9*4096 + 84	A word wrap field may not be right-justified.
FM-WRFILL	9*4096 + 85	A word wrap field may not be must-fill.
FM-WRNOTAB	9*4096 + 86	A word wrap field must allow auto-tab.
**Msgs 87-92 Free For Use		
FM-GRNONE	9*4096 + 93	Graphics not available on this terminal.
**Msgs 94-100 Free For Use		
FM-FLEN-EXSHF	9*4096 + 101	Field too long for maximum shifting length.
FM-GNOFLDS	9*4096 + 102	
FM-OVERLAP	9*4096 + 103	Overlaps field or border.
FM-NAMEINUSE	9*4096 + 104	Name already assigned to another field or group.
FM-FLDNO	9*4096 + 105	Invalid field name.
FM-CLCMIN	9*4096 + 110	Minimum digits should not exceed length of field, which is %d.
FM-LINES	9*4096 + 116	lines
FM-COLS	9*4096 + 117	columns

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-INBORDER	9*4096 + 118	Bad entry — field in prospective border.
FM-DUPDRAW	9*4096 + 120	Duplicate draw character.
Validation Bits For Summary Window		
FM-RTJUST	9*4096 + 121	"RT-JUST "
FM-REQD	9*4096 + 122	"REQUIRED "
FM-CLRINP	9*4096 + 123	"CLR-INPUT "
FM-MENU	9*4096 + 124	"MENU "
FM-RETURN	9*4096 + 125	"RETURN "
FM-UPPER	9*4096 + 126	"UPPER "
FM-LOWER	9*4096 + 127	"LOWER "
FM-FILLED	9*4096 + 128	"MUST-FILL "
FM-NO-AUTO	9*4096 + 129	"NO_AUTOTAB "
FM-WRAP	9*4096 + 130	"WORDWRAP "
FM-APROT	9*4096 + 131	"PROTECTED "
FM-EPROT	9*4096 + 132	"E-PROT "
FM-TPROT	9*4096 + 133	"T-PROT "
FM-CPROT	9*4096 + 134	"C-PROT "
FM-VPROT	9*4096 + 135	"V-PROT "
FM-IFORMAT	9*4096 + 136	Invalid format.
FM-INVRC	9*4096 + 137	Invalid menu return code.
FM-WRMSK	9*4096 + 138	A word wrap field may not have a field-level regular expression.

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
Regular Expressions		
FM-RX1	9*4096 + 139	Regular expression too long.
FM-RX2	9*4096 + 140	Unbalanced '[' bracket.
FM-RX3	9*4096 + 141	Too many '(' brackets.
FM-RX4	9*4096 + 142	Too many ')' brackets.
FM-RX5	9*4096 + 143	Expecting number between 0–9 or '\}'.
FM-RX6	9*4096 + 144	Range may not exceed 255.
FM-RX7	9*4096 + 145	Too many commas in specifying range.
FM-RX8	9*4096 + 146	Closing '}' brace expected.
FM-RX9	9*4096 + 147	First number exceeds second in specifying range.
FM-RX10	9*4096 + 148	digit out of range.
FM-RX11	9*4096 + 149	Previous '(' bracket not yet closed.
FM-RX12	9*4096 + 150	Unexpected end of regular expression.
FM-RX13	9*4096 + 151	Range can follow only an expression.
Data For FTYPE Window		
FM-FTOMIT	9*4096 + 159	omit from struct
FM-FTSTR	9*4096 + 160	<S>char string
FM-FT1	9*4096 + 161	int
FM-FTDIGIT	9*4096 + 162	unsigned int

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-FT3	9*4096 + 163	short int
FM-FT4	9*4096 + 164	long int
FM-FT5	9*4096 + 165	<R>float
FM-FTNUM	9*4096 + 166	<R>double
FM-FT7	9*4096 + 167	zoned decimal
FM-FT8	9*4096 + 168	packed decimal
FM-FT9	9*4096 + 169	no C default
FM-FT10	9*4096 + 170	no C default
FM-FT11	9*4096 + 171	no C default
FM-FT12	9*4096 + 172	no C default
FM-FT13	9*4096 + 173	no C default
Edits For Summary Window		
FM-RANGES	9*4096 + 174	"RANGES "
FM-NEXT	9*4096 + 175	"NEXT-FLD "
FM-CURRENCY	9*4096 + 176	"CURRENCY "
FM-TEXT	9*4096 + 177	"TEXT "
FM-VALPROG	9*4096 + 178	"VAL-FUNC "
FM-HELP	9*4096 + 179	"HELP "
FM-CALC	9*4096 + 180	"MATH "
FM-SDATETIME	9*4096 + 181	"SYST-DATE/TIME "
FM-FXPROG	9*4096 + 182	"FLD-EXIT-FUNC "
FM-CDIGIT	9*4096 + 183	"CK-DIGIT "

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-TYPE	9*4096 + 184	"TYPE "
FM-UDATETIME	9*4096 + 185	"USR-DATE/TIME "
FM-KSBADVAL	9*4096 + 186	"Bad value for soft key."
FM-ITEM	9*4096 + 187	"ITEM-SELECT "
FM-AHELP	9*4096 + 188	"AUTO-HELP "
FM-AITEM	9*4096 + 189	"AUTO-ITEM-SEL "
FM-MEMOS	9*4096 + 190	"MEMOS "
FM-FEPROG	9*4096 + 191	"FLD-ENTRY-FUNC "
FM-RETCODE	9*4096 + 192	"RET-CODE "
FM-JPLEDIT	9*4096 + 193	"JPL "
FM-SUBMENU	9*4096 + 194	"SUBMENU "
FM-REGEXP	9*4096 + 195	"REG-EXP "
FM-TBL-LOOKUP	9*4096 + 196	"TBL-LOOKUP "
FM-LONGDT	9*4096 + 197	Date/time format string too long
Currency Defaults		
FM-NOCURRDFLT	9*4096 + 198	WARNING: Currency default messages not found.
FM-0MN-CURRDEF	9*4096 + 199	CURRENCY DEFAULT
FM-1MN-CURRDEF	9*4096 + 200	NUMERIC DEFAULT
FM-2MN-CURRDEF	9*4096 + 201	PLAIN DEFAULT
FM-3MN-CURRDEF	9*4096 + 202	DEFAULT3
FM-4MN-CURRDEF	9*4096 + 203	DEFAULT4

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-5MN-CURRDEF	9*4096 + 204	DEFAULT5
FM-6MN-CURRDEF	9*4096 + 205	DEFAULT6
FM-7MN-CURRDEF	9*4096 + 206	DEFAULT7
FM-8MN-CURRDEF	9*4096 + 207	DEFAULT8
FM-9MN-CURRDEF	9*4096 + 208	DEFAULT9
FM-INV-CURRDFLT	9*4096 + 209	Invalid currency default.
NULL for Summary Window		
FM-NULLEDIT	9*4096 + 210	"NULL" — for summary window
Keyset Editor		
FM-KSSTATLN	9*4096 + 211	main status line
FM-KSNEWSIZE	9*4096 + 212	status line for new size.
FM-KSQSAVE	9*4096 + 213	Do you want to save the current key-set? (enter y or n)
FM-KSNOOPEN	9*4096 + 214	Cannot create keyset %s.
FM-KSSAVED	9*4096 + 215	keyset '%s' saved.
FM-KSWRITE	9*4096 + 216	Error writing keyset '%s'.
FM-KSTOOBIG	9*4096 + 217	Keyset has too many rows.
Date/Time Strings For Local Dialect.		
FM-YR4	9*4096 + 218	
FM-YR2	9*4096 + 219	
FM-MON	9*4096 + 220	
FM-MON2	9*4096 + 221	

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-DATE	9*4096 + 222	
FM-DATE2	9*4096 + 223	
FM-HOUR	9*4096 + 224	
FM-HOUR2	9*4096 + 225	
FM-MIN	9*4096 + 226	
FM-MIN2	9*4096 + 227	
FM-SEC	9*4096 + 228	
FM-SEC2	9*4096 + 229	
FM-YRDAY	9*4096 + 230	
FM-AMPM	9*4096 + 231	
FM-DAYA	9*4096 + 232	
FM-DAYL	9*4096 + 233	
FM-MONA	9*4096 + 234	
FM-MONL	9*4096 + 235	
Mnemonics To Specify Default Dates In the Screen Editor		
FM-0MN-DEF-DT	9*4096 + 236	DEFAULT
FM-1MN-DEF-DT	9*4096 + 237	DEFAULT1
FM-2MN-DEF-DT	9*4096 + 238	DEFAULT2
FM-3MN-DEF-DT	9*4096 + 239	DEFAULT3
FM-4MN-DEF-DT	9*4096 + 240	DEFAULT4
FM-5MN-DEF-DT	9*4096 + 241	DEFAULT5
FM-6MN-DEF-DT	9*4096 + 242	DEFAULT6

Screen Editor Messages: FM-MSGS = 9		
Mnemonic	*Message Number*	*Message Text*
FM-7MN-DEF-DT	9*4096 + 243	DEFAULT7
FM-8MN-DEF-DT	9*4096 + 244	DEFAULT8
FM-9MN-DEF-DT	9*4096 + 245	DEFAULT9
Clipboard Messages		
FM-BIGCLIP	9*4096 + 246	Clipboard too big to fit in screen
FM-EMPTYCLIP	9*4096 + 247	Clipboard '%c' is empty

JAM *Messages:* JM-MSGS = 10		
Mnemonic	*Message Number*	*Message Text*
JM-BYE	10*4096 + 0	\nThank you! Have a nice day.\n
JM-QTERMINATE	10*4096 + 1	"Would you like to terminate this session(y/n)? "
JM-BIGPARAM	10*4096 + 2	Parameter list is to big
JM-OVFORM	10*4096 + 3	Form stack overflow
JM-LONGNAME	10*4096 + 4	Screen name '%s' is too long
JM-HITSPACE	10*4096 + 5	Hit space bar to continue
JM-NDXREBUILD	10*4096 + 6	Rebuilding index..
JM-NDXINIT	10*4096 + 7	Initializing index..
JM-INVENTRY	10*4096 + 10	Invalid entry
LDB Messages		
JM-NODD	10*4096 + 11	No Data Dictionary file.
JM-NONDX	10*4096 + 12	Index not initialized.

JAM *Messages:* JM-MSGS = 10		
Mnemonic	*Message Number*	*Message Text*
JM-INVFILE	10*4096 + 14	Error: initialization file %s is invalid
JM-BIGNAME	10*4096 + 15	Warning: name %s too long
JM-NOITEM	10*4096 + 16	Item %s does not exist in Data Dictionary
JM-BIGELE	10*4096 + 17	Warning: element number %d exceeds occurrences for %s\n
JM-BIGINIT	10*4096 + 18	Warning: init string for %s too long; truncated
JM-BADDATA	10*4096 + 19	Warning: bad data, no %s initialization
JM-WAIT	10*4096 + 20	Please wait..
JM-NOINI	10*4096 + 21	Warning: Initiallization file %s not found.
JM-HITACK	10*4096 + 22	Hit ACKnowledge key to continue
JM-NOMEMDD	10*4096 + 23	Not enough memory for Data Dictionary.
JM-DDINV	10*4096 + 24	Invalid Data Dictionary.
JM-BIGINIT2	10*4096 + 25	Warning: init string for %s occ# %d too long; truncated
JM-HITFUNC	10*4096 + 26	Hit any function key to continue.

JXFORM MESSAGES: JX-MSGS = 11*		
Mnemonic	*Message Number*	*Message Text*
JX-NOCINIT	11*4096 + 0	Warning: no constants initialization file
JX-EMPCINIT	11*4096 + 1	Warning: constants initialization file empty
JX-EMPGINIT	11*4096 + 3	Warning: globals initialization file empty
JX-PROGNAME	11*4096 + 5	JXFORM Rel X.X ...
JX-USAGE	11*4096 + 6	Usage: jxform [-e] [screen ...]\n
JX-EDIT-OPT	11*4096 + 7	Start jxform in edit mode
JX-DESCR	11*4096 + 8	Displays and modifies JAM screens.
JX-SHOWFLD	11*4096 + 4	show field names
JX-SHOWGRP	11*4096 + 84	show group names
JX-JCSEXISTS	11*4096 + 10	'"%s" field already exists.'
JX-MISDD	11*4096 + 11	Data Dictionary not found.
JX-DDREAD	11*4096 + 12	Reading Data Dictionary File
JX-NOTFIELD	11*4096 + 13	Cursor is not in a field.
JX-NONAME	11*4096 + 14	Field has no name.
JX-ENTEXIST	11*4096 + 15	Entry already exists.
JX-DDDATA	11*4096 + 16	Bad data in data dictionary file.
JX-DDLIMIT	11*4096 + 17	Cannot update; new element count exceeds %d limit.
JX-DDEREAD	11*4096 + 18	Error reading data dictionary file
JX-DDCREATE	11*4096 + 19	Cannot create data dictionary.

JXFORM MESSAGES: JX-MSGS = 11*		
Mnemonic	*Message Number*	*Message Text*
JX-DDUPDATE	11*4096 + 20	Cannot update data dictionary.
JX-DDWRITE	11*4096 + 21	Failure in writing data dictionary
JX-ADDENTRY	11*4096 + 22	Added entry "%s"
JX-NOUPDATE	11*4096 + 30	Can't update file. LATEST is '%s'; '%s' is backup
JX-ARESURE	11*4096 + 31	"Are you sure (y/n)? "
JX-NOREBUILD	11*4096 + 33	Cannot rebuild index.
JX-DDSAVED	11*4096 + 34	Data Dictionary saved.
JX-NORECORD	11*4096 + 39	obsolete mnemonic)
JX-NODELETE	11*4096 + 39	No entry deleted or already undeleted.
JX-NOROOM	11*4096 + 40	No room in data dictionary.
JX-WRONGTERM	11*4096 + 41	Cannot perform requested function on this terminal
JX-TMPOPEN	11*4096 + 42	Can't open %s; try writing %s.
JX-OPENFATAL	11*4096 + 43	Can't open %s; please exit DD editor.
JX-WRITING	11*4096 + 44	Writing Data Dictionary File..
JX-LGOPEN	11*4096 + 45	Can't write %s; delete some items and try again.
JX-READ	11*4096 + 46	Cannot read %s.
JX-BADDATA	11*4096 + 47	Bad data in %s.
**Msgs 48 Free For Use		
**Msgs 60 – 63 Free For Use		

JXFORM MESSAGES: JX-MSGS = 11*		
Mnemonic	*Message Number*	*Message Text*
JX-ITMEXIST	11*4096 + 64	obsolete mnemonic)
JX-RECEXIST	11*4096 + 64	Record already exists.
JX-NOFIELDS	11*4096 + 65	Data dictionary has no fields
JX-NOGROUPS	11*4096 + 66	Data dictionary has no groups
JX-ITMNOTFOUND	11*4096 + 67	Item not found.
JX-NOSEARCH	11*4096 + 68	Don't know what to search.
Status Lines		
JX-0STLINE	11*4096 + 70	JAM DRAW mode status line
JX-1STLINE	11*4096 + 71	JAM TEST mode status line
JX-4STLINE	11*4096 + 72	basic JXFORM status line
JX-5STLINE	11*4096 + 73	DD editor status line
JX-6STLINE	11*4096 + 74	DD find match status line
JX-7STLINE	11*4096 + 75	DD editor update status line
JX-8STLINE	11*4096 + 76	DD template status line
JX-9STLINE	11*4096 + 77	DD/field match status line
JX-10STLINE	11*4096 + 78	base line without keyset ed
**MOVE/COPY Use Screen Editor Status Lines		

* jxform messages other than those listed under *Screen Editor* & **JAM.**

Addendum

for Updates to JAM Release 5.03

for Stratus COBOL

Part Number R333-00A

August 3, 1992

Note of Explanation

This addendum describes new features in release 5.03 of JAM. This addendum is for the Stratus COBOL *Programmer's Guide*. There are separate addenda for Volumes 1 and 2 of the JAM 5.03 documentation set.

Several insertion pages (or A–pages) are included for new library routines and utilities in JAM 5.03. These pages should be inserted into your JAM *Programmer's Guide* and *Utilities Guide* at the appropriate location. For example, page A–195 should be inserted before page 195.

Note that the page numbers for the *Utilities Guide* refer to the August 1, 1991 printing of the JAM manual. Page numbers in the *Programmer's Guide* refer to the March 1, 1991 printing of the Stratus COBOL *Programmer's Guide*.

Stratus COBOL **Programmer's Guide**

Page 94: New Behavior and Return Codes for `xsm_ascroll`

The library routine `xsm_ascroll` takes as arguments a field number and an occurrence. It scrolls an array such that the requested occurrence is in the specified field. If the requested occurrence cannot be placed in the specified field because it is one of the first or last occurrences in a non–circular array, then `xsm_ascroll` scrolls the occurrence onto the screen and returns the occurrence number of the occurrence that is actually in the specified field.

Page 170: Inquiring Help Level via `xsm_inquire`

The global variable `I_INHELP` now contains the level of help that the user is in, instead of just a true/false value. There may be up to five levels of help. Use `sm_inquire` to query the value of this variable. A return of zero indicates that the user is not in help, a return of 1 through 5 indicates which help level the user is in.

Page 197: `xsm_keyoption`

Certain keys can not be translated via the `KEY_XLATE` argument to `sm_keyoption`. These are: INS, REFR, SFTS, LP, and ABORT. They may, however, be disabled via the `KEY_ROUTING` argument, or intercepted via a keychange function

Page 224: `xsm_msgread`

The header file `msgfile.incl.cobol` is a user–created file that is necessary only if you are using a memory–resident message file.

Page 249: Percent Escapes in `xsm_query_msg`

Percent escapes are now supported for controlling the attributes of query messages. The sequences are the same as those for `xsm_emsg`, and detailed on page 214. Note that `%Mu` and `%Md` are not supported. Query messages from JPL can also now use percent escapes.

Page 295: MDT bits and Scrolling Arrays

When lines are inserted or deleted from scrolling arrays via INSL or DELL, the MDT bits for all occurrences after the insertion or deletion are no longer set. In a database application, this prevents the need for unnecessary processing to write potentially large amounts data that have not changed. For large arrays, it can save a significant amount of processing time.

bin2cob

convert binary **JAM** files to COBOL copy files

SYNOPSIS

`bin2cob [-fv]` ***COBOL–file binary–file...***

OPTIONS

`-f` Overwrite an existing output file.

`-v` Generate list of files processed.

DESCRIPTION

This program converts binary files created with other **JAM** utilities into COBOL source. ***COBOL–file*** is usually a new file name. (To overwrite an existing file, you must use the `-f` option.)

When the utility creates the COBOL source file, it generates a copy file for each of the binary input files. The name of the copy file is derived from the binary file name, with the path and extension removed, and given the extension `.incl.cobol`. Each copy file contains one `01` level with the name of the binary file followed by `-form`. Under that there are multiple `05` levels, all named `filler`, that are given initial values representing the data in the binary file.

The application program should include the copy file in the program that uses it. The `_d` variants of certain library routines (`d_window`, `d_form`, `d_at_cur`, `d_keyset`, `d_msg_line`) can then be used.

`bin2cob` copy files may be compiled, linked with your application, and added to the memory–resident form list. (See the **JAM** *Programmer's Guide* for more information on memory–resident lists.) The following files may be made memory–resident:

- key translation files (`key2bin`)
- setup variable files (`var2bin`)
- video configuration files (`vid2bin`)
- message files (`msg2bin`)
- JPL files (`jpl2bin`)
- screen files (`jxform`)

There is no utility to convert ***ascii–file*** back to its original binary form after using `bin2cob`. **JAM** provides other utilities that permit two–way conversions between binary and ASCII formats. For screens, these utilities are `bin2hex` and `f2asc`.

ERRORS

`Insufficient memory available.`
Cause: The utility could not allocate enough memory for its needs.
Corrective action: Try to increase the amount of available memory.

`File "%s" already exists; use '-f' to overwrite.`
Cause: You have specified an output file that already exists.
Corrective action: Use the –f flag to overwrite the file, or use another name.

`"%s": Permission denied.`
Cause: An input file was not readable, or an output file was not writeable.
Corrective action: Check the permissions of the file in question.

copyarray

copy the contents of one array to another

SYNOPSIS

```
77 destination-fld    pic S(9)9 comp-5.
77 source-fld         pic S(9)9 comp-5.
77 status             pic S(9)9 comp-5.
call "xsm_copyarray" using destination-fld, source-fld giving
      status.
```

DESCRIPTION

This routine copies the contents of the array containing `source-fld` into the array containing `destination-fld`. `source-fld` and `destination-fld` are field numbers. They may be the field number of any of element in the respective array.

The developer is responsible for insuring that the arrays are compatible. Data in source array occurrences that are too long for the destination array are truncated without warning. Data in source array occurrences that are shorter than the destination array's field length are blank filled (with respect for justification).

If the source array has more occurrences than the destination array, the data in the extra occurrences are discarded. If the source array has fewer occurrences than the destination array, trailing occurrences in the destination array are cleared of data (but not de-allocated).

`copyarray` sets the MDT bit and clears the VALIDED bit for each destination array occurrence, indicating that the occurrence has been modified and requires validation.

The variant, `xsm_n_copyarray`, searches the LDB for either array if the named field is not found on the screen. However, if the destination LDB item has a scope of 1, meaning that it is a constant, then it is not altered and the function returns –1. .

RETURNS

–1 if either field is not found or if the destination array in the LDB has a scope of 1.
0 otherwise.

VARIANTS

```
call "xsm_n_copyarray" using destination-name, source-name
      giving status.
```

RELATED FUNCTIONS

```
call "xsm_clear_array" using field-number giving status.
```

```
call "xsm_getfield" using buffer, field-number giving length.
call "xsm_putfield" using field-number, data giving status.
```

next_sync

find next synchronized array

SYNOPSIS

```
77 field_number       pic S(9)9 comp-5.
77 next-array         pic S(9)9 comp-5.
call "xsm_next_sync" using field-number giving next-array.
```

DESCRIPTION

Given a field number, this function finds the next array synchronized with the given field, and returns the field number of the corresponding element in that array. The next synchronized array is defined as the one to the right. If `field-number` is in the rightmost synchronized array, the function returns the corresponding element in the leftmost synchronized array (ie– it wraps around the screen).

RETURNS

The field number of the corresponding element in the next synchronized array if there is one.
Otherwise, the field number the function was passed.

soption

set a string option

SYNOPSIS

```
copy "smmisc.incl.cobol".
copy "smmisc2.incl.cobol".
copy "smmisc3.incl.cobol".

77 option              pic S(9)9 comp-5.
77 newval              display-2 pic x(256).
77 oldval              display-2 pic x(256).
call "xsm_soption" using option, newval GIVING OLDVAL.
```

DESCRIPTION

Use `xsm_soption` to alter during run-time the default string options defined in `smsetup.incl.cobol`. The following table lists the valid values for `option`:

Value	*Description*
`SO_EDITOR`	Editor to use in JPL windows.
`SO_FEXTENSION`	Screen file extension.
`SO_LPRINT`	Operating system print command.
`SO_PATH`	Search path for screens and JPL procedures.

These variables are fully documented in the JAM *Configuration Guide*, under "System Environment and Setup Files."

RETURNS

The old value for the specified option.
0 if the option is invalid or a malloc error occurred.

RELATED FUNCTIONS

```
call "xsm_option" using option, newval GIVING OLDVAL.
```

wrotate

rotate the display of sibling windows

SYNOPSIS

```
77 step                pic S(9)9 comp-5.
77 status              pic S(9)9 comp-5.
call "xsm_wrotate" using step giving status.
```

DESCRIPTION

If two or more sibling windows are on the top of the display, this function may be used to rotate the sequence of the sibling windows. `step` is a positive or negative integer equalling the number of screen rotations. If `step` is positive, the routine takes the top-most sibling window and makes it the last sibling window for each instance of `step`. If `step` is negative, the routine takes the last sibling window and makes it first. If `step` is zero, no rotations are performed. See the figures below.

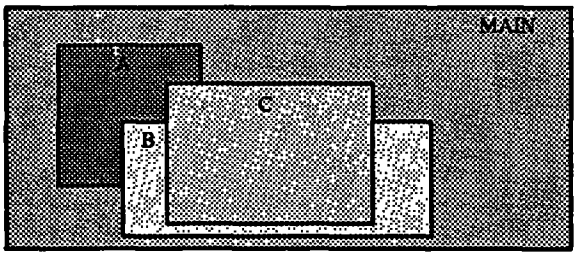


Figure 1: Screens `a`, `b`.and `c` are all siblings.Screen `main` is not a sibling.

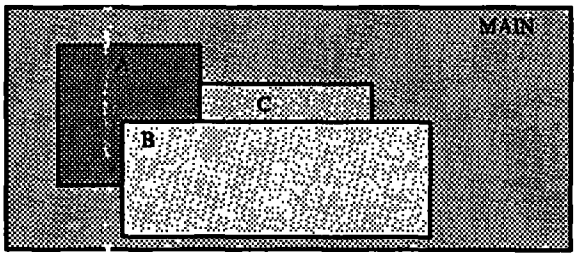


Figure 2: Executing `sm_wrotate (1)` rotates the top sibling to the bottom of the sibling stack. It rotates screen `c` behind the other two sibling windows, leaving screen `b` on top. Screen `main` is not affected.

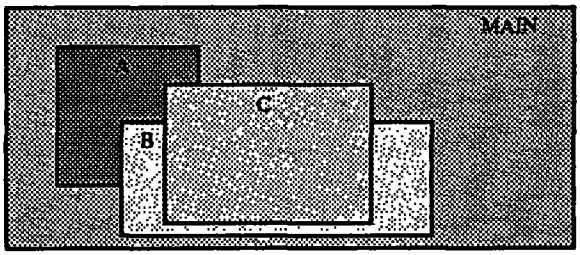


Figure 3: Executing `sm_wrotate (-1)` rotates the last sibling window to the top, putting screen `c` on top. The display is the same as Figure 1.

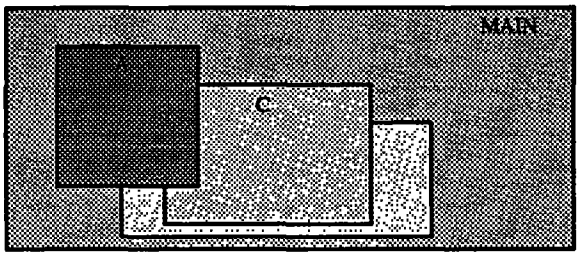


Figure 4: Executing `sm_wrotate (2)` rotates the first two sibling windows off the the top. First it rotates screen `c` to the back; then screen `b`, leaving screen `a` on top.

RETURNS

One less than the number of sibling windows on top of the window stack.
0 if there are no sibling windows

RELATED FUNCTIONS

```
call "xsm_sibling" using should-it-be.
```