

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.

JAM is a registered trademark and JAM/Presentation interface is a trademark of JYACC, Inc.

IBM, PC/XT, IBM AT, PS/2, and IBM PC are registered trademarks of International Business Machines Corporation.

Windows is a trademark and Microsoft, MS, and MS-DOS are registered trademarks of Microsoft Corporation.

The X Window System is a trademark of the Massachusetts Institute of Technology.

OSF/Motif is a trademark of the Open Software Foundation.

UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc.

Helvetica and Times are registered trademarks of Linotype Company.

Times Roman is a registered trademark of Monotype Corporation.

Other product names mentioned in this manual may be trademarks of their respective proprietors, and they are used for identification purposes only.

Please send suggestions and comments regarding this document to:

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

(212) 267-7722

© 1992 JYACC, Inc.
All rights reserved.
Printed in USA.

TABLE OF CONTENTS

Chapter 1		
Introduction	1
1.1	About This Document	1
1.1.1	Conventions	1
1.2	What is the JAM/Presentation interface?	2
1.3	Using JAM/Pi Effectively	2
1.4	Overview of Features in JAM/Pi	4
1.4.1	Portability Across Environments	4
1.4.2	Compatibility with Character JAM	5
1.4.3	Support for GUI features	5
	Transformation of Objects and Text	5
	Extended Functionality	5
	Extended Fonts and Colors	5
	Application Defaults	7
	Extended Library Routines	8
 Chapter 2		
JAM Objects into GUI Widgets	11
2.1	Introduction	11
2.2	Widget Attributes	12
2.2.1	Widget Attribute Hierarchy	12
2.2.2	Application-Wide Attributes	13
2.2.3	Screen-Wide Attributes	15
2.2.4	Widget-Specific Attributes	16
2.3	Transformation into Widgets	17
2.3.1	Display Text and Protected Fields	17
2.3.2	Data Entry Fields	17
2.3.3	Arrays	18
2.3.4	Menus	19
2.3.5	Groups	20

Chapter 3	
Arranging Screens in JAM/Pi	23
3.1 Overview of Positioning	23
3.2 Anchoring	26
3.2.1 Anchoring by Field Justification	26
3.2.2 Horizontal Anchoring: the halign Field Extension	26
3.2.3 Vertical Anchoring: the valign Field Extension	27
3.2.4 Anchoring Display Text	28
3.3 Whitespace	29
3.4 Proportional vs. Fixed Width Fonts	30
3.5 Widget Size	32
3.6 Fine Tuning Screen Arrangement	33
3.6.1 The space Field Extension	33
3.6.2 The noadj Field Extension	33
3.6.3 The hoff and voff Field Extensions	34
3.7 Refreshing the Screen	35
3.8 Separator Rows and Columns	36
3.8.1 Separators and the Elastic Grid	37
 Chapter 4	
JAM Behavior in a GUI Environment	39
4.1 JAM Screens	39
4.1.1 Title Bars	39
4.1.2 Multiple Document Interface in MS Windows	40
4.1.3 Focus	41
4.1.4 JAM Borders	42
4.1.5 Iconification	43
Preventing Iconification	43
4.1.6 Toggling Between Menu Mode and Data Entry Mode	44
4.2 Error and Status Messages	44
4.2.1 Dialog Box Icons	46
4.2.2 Location of the Status Line	47
Status Line Keytops	47
Keytop Functions in the Authoring Tool	47

4.3	Shifting and Scrolling	47
4.3.1	Shifting Fields and Proportional Fonts	48
4.3.2	User Interface to Shifting and Scrolling	49
4.3.3	Shifting and Scrolling Indicators	49
	Turning Off JAM Shift/Scroll Indicators	49
	Changing the Characters Used as Indicators	50
4.4	Cutting, Copying & Pasting Text	50
4.5	Soft Keys	51
4.5.1	Location of Soft Keys	52
4.5.2	Soft Keys vs. Menu Bars	52
	The kset2mnu Utility	52

Chapter 5		
Entering Screen and Field Extensions	53	
5.1	Introduction	53
5.2	The Screen Extensions Window	54
5.2.1	The Details Window for Lines and Boxes	58
5.3	The Field Extensions Window	61
5.3.1	Synchronizing JAM and the GUI	61
5.3.2	Forcing the Widget Type	61
5.3.3	Entering Data in the Field Extensions Window	62
5.3.4	The Frame Window	66
5.3.5	Widget Details Windows	68
5.3.6	The Size and Alignment Window	71

Chapter 6		
Extension Reference	75	
6.1	Introduction	75
6.2	Extension Syntax	76
6.2.1	Colon Expansion of Extension Arguments	76
6.3	Propagating Extensions	77
6.4	Extension Reference	77
bg		
fg	specify background or foreground color for a screen or widget .	81
box	draw a box	84
checkbox	create a checklist style toggle button	87
dialog	create a dialog box from a screen	88
font	specify the font for a screen or widget	89

frame	create a frame around a widget	92
halign		
valign	specify alternative horizontal or vertical alignment for a widget	94
height		
width	specify the width or height of a widget	96
hline		
vline	create a vertical or horizontal line	98
hoff		
voff	specify a horizontal or vertical offset for a widget	102
icon	enable iconification and associate an icon with a screen	104
iconify	start this screen as an icon	106
label	create a label widget	107
list	create a list box from an array	108
maximize	invoke a window maximized	110
multiline	create a multiline label for a menu or group button	111
multitext	create a multiline text widget from an array	113
noadj	disable vertical or horizontal grid adjustment for a widget	115
noborder	suppress the GUI border for this screen	116
noclose	suppress the close option on the GUI window menu	118
nomaximize	prevent the user from maximizing a window	119
nomenu	suppress the GUI window menu	120
nominimize	prevent the user from minimizing a GUI window	122
nomove	suppress the move option on the GUI window menu	123
noresize	prevent the user from resizing a GUI window	124
notitle	suppress title bar	125
nowidget	don't create a GUI widget for this field	126
optionmenu	create an option menu widget	127
pixmap	associate a bitmap or pixmap with a label	130
pointer	specify the pointer shape	134
pushbutton	create a pushbutton widget	136
radiobutton	create a radio style toggle button	138
scale	create a scale widget	139
space	equally space the elements of an array	140
text	create a text widget	141
title	change the title bar on a screen	142
togglebutton	create an in/out style toggle button	143

Chapter 7		
Setting Application Defaults		145
7.1	Resource and Initialization Files	145
7.1.1	Resource and Initialization File Names	145
7.1.2	Structure of Resource and Initialization Files	146
7.1.3	Location of Resource and Initialization Files	148
7.2	Colors	149
7.2.1	Setting JAM Palette Colors	149
7.2.2	Colors Beyond the JAM Palette	151
	Motif Color Resources	151
	OPEN LOOK Color Resources	151
	Motif/OPEN LOOK Background and Foreground Resources	152
7.3	Fonts	153
7.3.1	Where Fonts are Specified	153
	The Application Default Font	153
	The Default Screen Font	154
	A Widget's Font	154
7.3.2	Naming Fonts	155
	Windows font naming	155
	Motif and OPEN LOOK font naming	155
7.4	Aliasing: GUI Independent Fonts and Colors	158
	Restrictions on Aliasing	159
7.5	Windows Initialization Options	160
7.5.1	The [Jam Options] Section of the Initialization File	160
	GrayOutBackgroundForms	160
	FrameTitle	160
	StartupSize	160
	StatusLineColor	161
	SMTERM	161
7.5.2	The Windows Control Panel and win.ini File	161
7.5.3	Highlighted Background Colors in Windows	161
7.5.4	Sample jam.ini File	162
7.6	Motif and OPEN LOOK Common Resource Options	163
7.6.1	Motif and OPEN LOOK Behavioral Resources	163
	The baseWindow Resource	163
	The formStatus Resource	163
	The formMenus Resource	164
	Combinations of baseWindow, formMenus and formStatus	164

7.6.2	Restricted Resources	165
7.6.3	Suggested Resource Settings	165
7.6.4	The rgb.txt File in Motif and OPEN LOOK	166
7.7	Motif Resource Options	167
7.7.1	Motif Global Resource and Command Line Options	167
7.7.2	Widget Hierarchy in Pi/Motif	168
	Base Screen	168
	Dialog Boxes	169
	JAM Screens	169
	Fields	171
	Display Text, Lines and Boxes	173
	Menu Bars	173
7.7.3	Sample Motif Resource File for JAM	175
7.8	OPEN LOOK Resource Options	179
7.8.1	OPEN LOOK Global Resource and Command Line Options ..	179
7.8.2	The OPEN LOOK keepOnScreen Resource	180
7.8.3	Widget Hierarchy in Pi/OPEN LOOK	180
	Base Screen	181
	JAM Screens	182
	Dialog Boxes	183
	Fields	183
	Display Text, Lines and Boxes	185
	Menu Bars	185
7.8.4	Sample OPEN LOOK Resource File for JAM	188

Chapter 8		
Menu Bars		191
8.1	Introduction	191
8.2	Location of Menu Bars	191
8.2.1	Pop-Up Menu Bar in Motif and OPEN LOOK	192
8.3	Menu Bar Scope	192
8.4	The Menu Script	194
8.4.1	Menu Script Structure	194
8.4.2	Menu Script Components	194
8.4.3	Sample Menu Script	198
8.5	Testing Menu Bars in The Authoring Utility	200
8.6	Menu Bar Library Routines	201
	Prototyping Menu Bar Library Routines	202

8.7	Installing Menu Bars	202
8.7.1	Enabling Menu Bars	202
8.7.2	Installing Menu Bars of Various Scopes	202
	Installing an Application-Level Menu Bar	202
	Installing a Screen-Level Menu Bar	202
	Installing Override-Level Menu Bars	203
	Installing Memory-Resident Menu Bars	203
	Installing the System-Level Menu Bar	203
8.7.3	Storing a Menu Bar in Memory	203
8.8	Using Menu Bars Effectively	203
8.9	Menu Bars vs. Soft Keys	204
8.9.1	Using Libraries to Store Menu Bars and Keysets	204
8.9.2	Converting Keysets into Menu Bars	205

Chapter 9

Using the Mouse

9.1	Introduction	207
9.1.1	Mouse Cursor Display	207
9.1.2	Mouse Buttons	208
9.1.3	Mouse Functions	208
	Menu Bars	209
	Focus	209
	Move, Offset and Resize	210
	Moving the Cursor and Making Selections	210
	Scrolling and Shifting	211
	Editing Text	212
	Select Mode	212
	Miscellaneous	212

Chapter 10

GUI Specific Features

10.1	Overstrike Mode in Pi/Motif and Pi/OPEN LOOK	213
10.2	Interfacing with the GUI Library	213
10.3	System Commands in Pi/Windows	214

Chapter 11

Conversion Issues

11.1	Background Highlights	215
------	-----------------------------	-----

11.2	Line Drawing	215
11.3	JAM Version 4 Applications	216
11.4	JAM Version 5 Applications	216

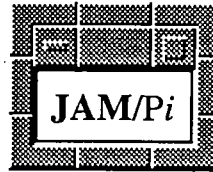
Chapter 12

	Library and Utility Reference	217
12.1	JAM/Pi Library Routines	217
	GUI Library Interface Routines	217
	Menu Bar Routines	217
	File Selection Box Routines	218
	Miscellaneous Routines	218
	sm_adjust_area refresh the current screen	219
	sm_c_menu close a menu bar	220
	sm_d_menu display a menu bar stored in memory	222
	sm_drawingarea get the widget id of the current JAM screen	224
	sm_filebox open a file selection dialog box	226
	sm_filetypes set up a list of file types for a file selection dialog box ..	231
	sm_menuinit initialize menu bar support	233
	sm_mn_forms install menu bars in memory	234
	sm_mnadd add an item to the end of a menu bar	235
	sm_mnchange alter a menu bar item	238
	sm_mndelete delete a menu bar item	240
	sm_mnget get menu bar item information	242
	sm_mninsert insert a new menu bar item	244
	sm_mnitems get the number of items on a menu bar	246
	sm_mnnew create a new menu bar by name	248
	sm_r_menu read & display a menu bar from memory, library or disk	250
	sm_translatecoords translate screen coordinates to display coordinates	252
	sm_widget get the widget id of a widget	255
	sm_win_shrink trim the current screen	257
12.2	Utilities	258
	menu2bin convert ASCII menu scripts to binary format	259
	kset2mnu convert keysets into ASCII menu scripts	261

Appendix A

	Terminology	263
	General Terms	263
	Terms Relating to Screens	264
	Terms Relating to Items on Screens	264

	Index	267
--	--------------------	------------



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.



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/Pi) product line provides a layer between the user and the application that enables JAM to support a variety of textual and graphical environments. JAM/Pi products include:

- JAM/Presentation *interface* for Microsoft Windows (Pi/Windows)
- JAM/Presentation *interface* for Motif (Pi/Motif)
- JAM/Presentation *interface* for OPEN LOOK (Pi/OPEN LOOK)
- JAM/Presentation *interface* for Graphics (Pi/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. Pi/Graphics is covered in a separate document. The abbreviation JAM/Pi, when used here, encompasses Pi/Windows, Pi/Motif and Pi/OPEN LOOK, but not Pi/Graphics.

The JAM/Pi 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/Pi 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/Pi.

The previous paragraph should not be taken to imply that JAM/Pi applications only *look like* GUI applications. In fact, applications developed with JAM/Pi *are* GUI compliant applications.

1.3

USING JAM/Pi EFFECTIVELY

In order to effectively use JAM/Pi, you must have an understanding of JAM. JAM screens are built from JAM objects: fields, groups, menus and display text. JAM ap-

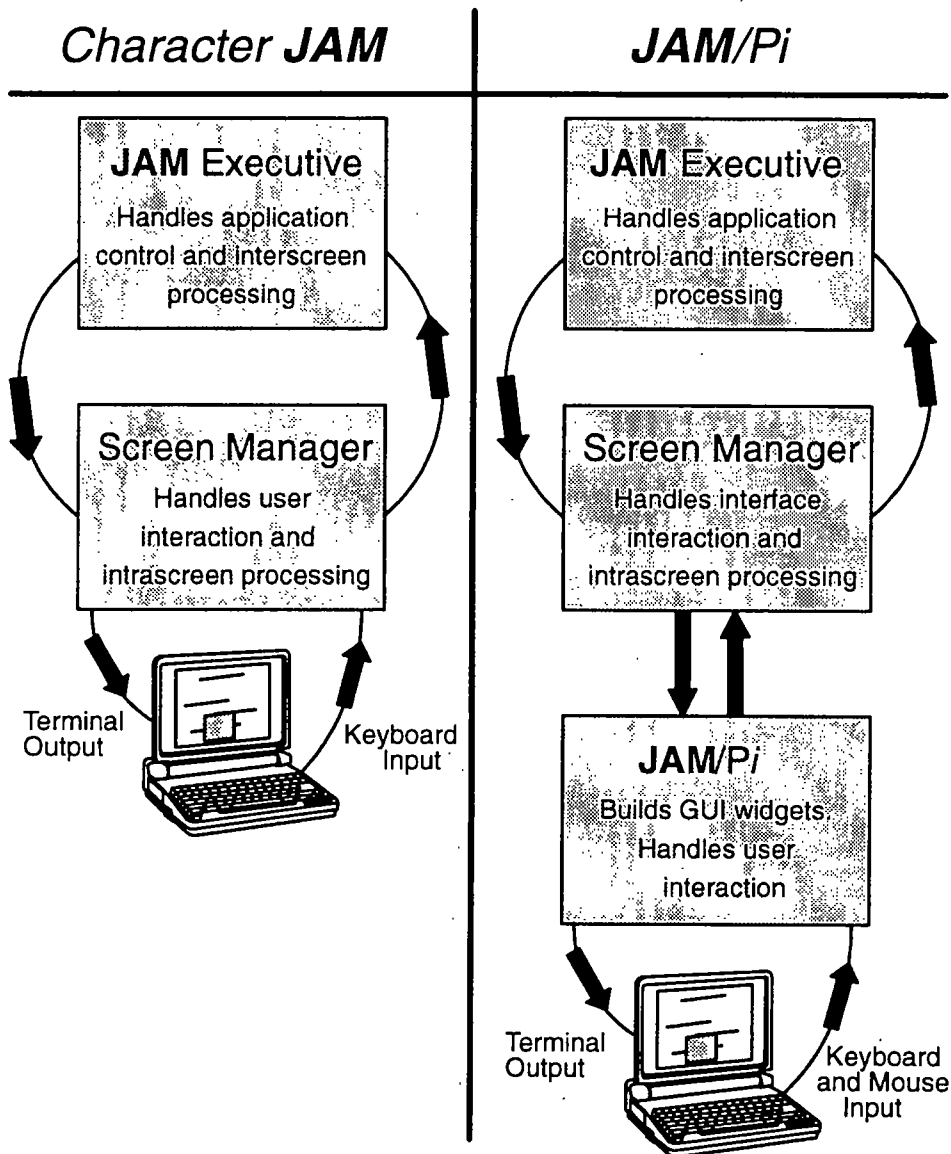


Figure 1: Schematic models of character **JAM** and **JAM/Pi**. User input, terminal output, and screen appearance are handled by the Presentation *interface* layer instead of the Screen Manager.

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/Pi** 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/Pi** applications are purely GUI based.

1.4.3

Support for GUI features

In order to create real GUI applications, **JAM/Pi** 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 **Pi/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/Pi** 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,

```
EMPLOYEE BENEFITS

401K Plan
Insurance
Childcare
Exit

NAME:Cindi_Phonemail

OPTIONS:

Principal Only
Dependents Only
X Principal/Dependents

Select a benefit category
```

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.

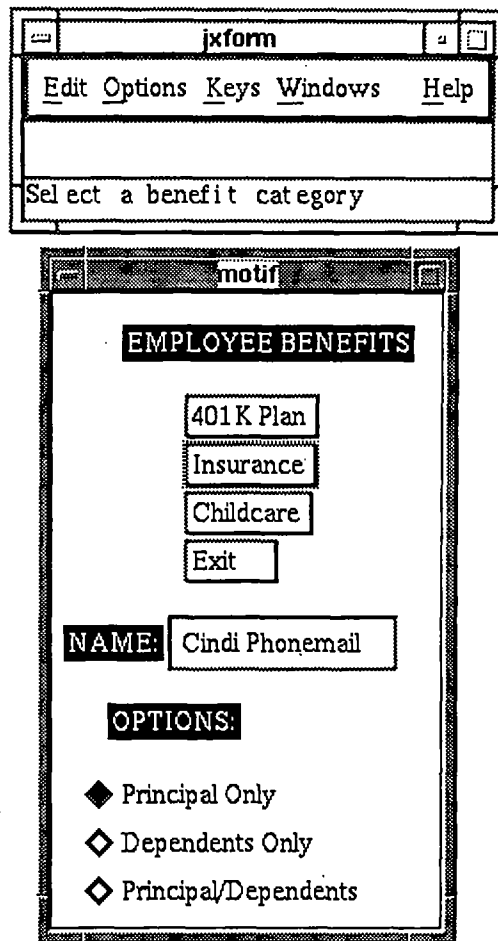


Figure 3: Same screen in P/Motif.

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.

401K PLAN OPTIONS

Percentage of weekly paycheck contributed: 6%

Instrument	Pct.
<input checked="" type="checkbox"/> Money Market	<u>17%</u>
<input type="checkbox"/> Growth Fund	<u>0%</u>
<input checked="" type="checkbox"/> Income Fund	<u>43%</u>
<input checked="" type="checkbox"/> Bond Fund	<u>40%</u>

Total Pct.: 100%

Update File View Emp History TRANSMIT EXIT

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.

401K Plan Options

Edit Windows Employee Keys

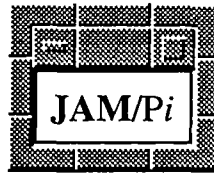
Update Data File

View Employee History

Percentage of weekly paycheck contributed %

Instrument	Pct.
<input checked="" type="checkbox"/> Money Market	<input type="text" value="17"/> %
<input type="checkbox"/> Growth Fund	<input type="text"/> %
<input checked="" type="checkbox"/> Income Fund	<input type="text" value="43"/> %
<input checked="" type="checkbox"/> Bond Fund	<input type="text" value="40"/> %
Total Pct.	<input type="text" value="100"/> %

Figure 5: P/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/Pi**. 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/Pi**, **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.

<i>JAM Object</i>	<i>Default Widget</i>
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 inherited from **JAM**. **JAM/Pi** extensions may be used to override these inherited 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.

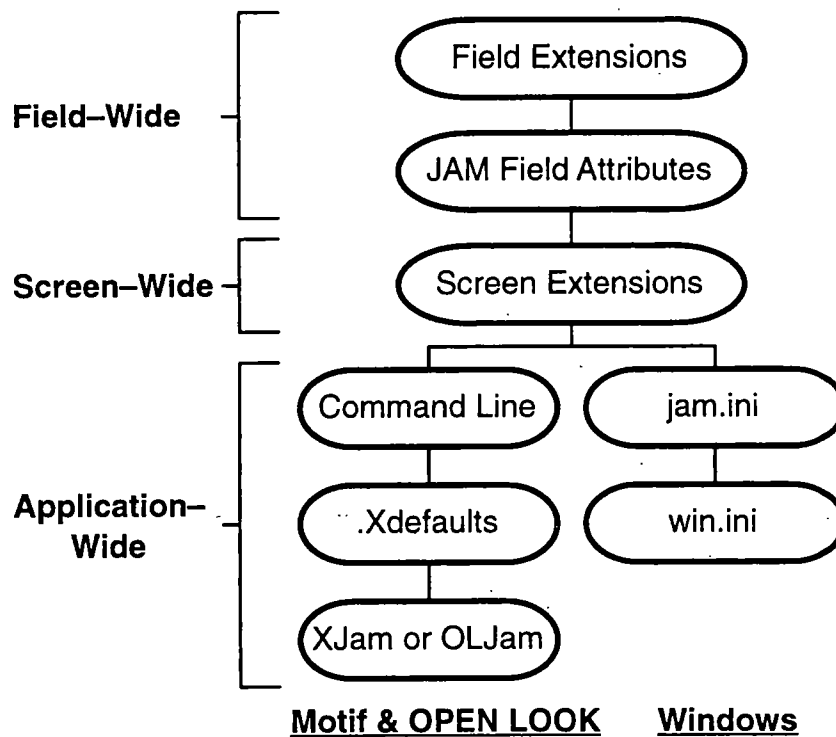


Figure 6: Hierarchy for widget attributes. Field extensions override screen extensions, which override the command line, etc.

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 Pi/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 Pi/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 Pi/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 Pi/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 Pi/Motif and Pi/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 Pi/Motif and Pi/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/Pi. Regions of display text are not fields, and therefore cannot have field extensions. They do however have JAM display attributes, and can inherit 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.

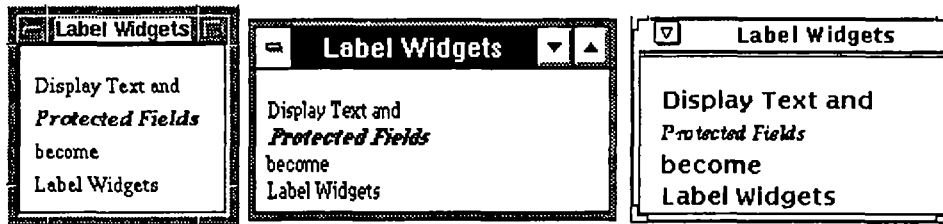


Figure 7: Label widgets in P//Motif, P//Windows and P//OPEN LOOK.

2.3.2

Data Entry Fields

Data entry fields become text widgets in JAM/Pi. The look and feel of the text widget is determined by the GUI, but the JAM field edits control its behavior.

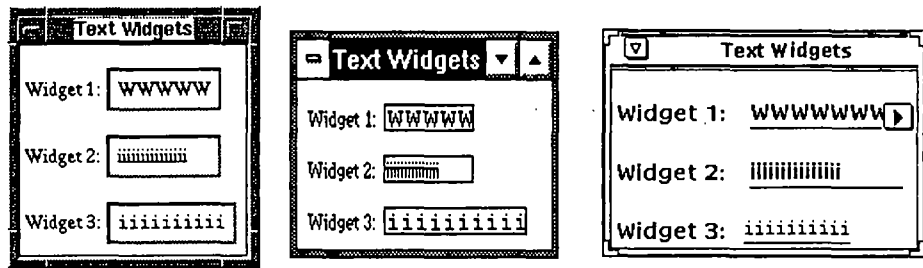


Figure 8: Text widgets in P//Motif, P//Windows and P//OPEN LOOK.

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.

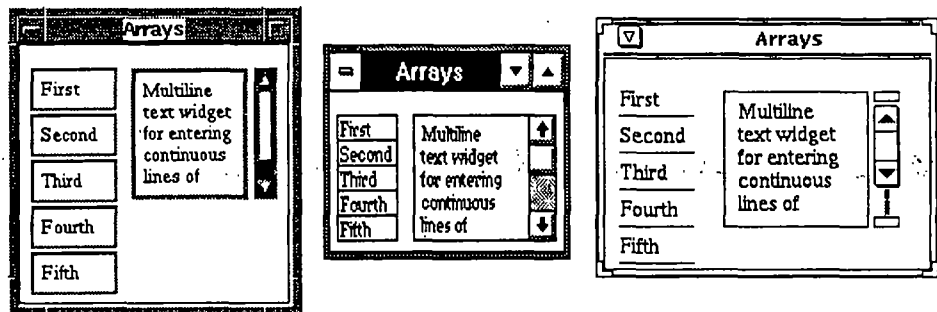


Figure 9: An array and multiline text widget in P//Motif, P//Windows and P//OPEN LOOK.

2.3.4

Menus

Menu fields appear as push buttons in JAM/Pi. 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.

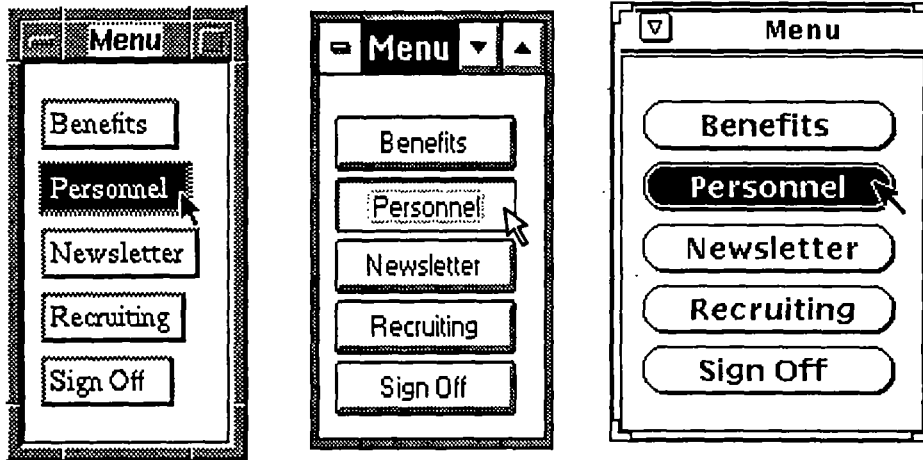


Figure 10: A set of menu fields in Pi/Motif, Pi/Windows and Pi/OPEN LOOK. The “Personnel” option is selected.



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/Pi 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*area.oblongButton.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//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.

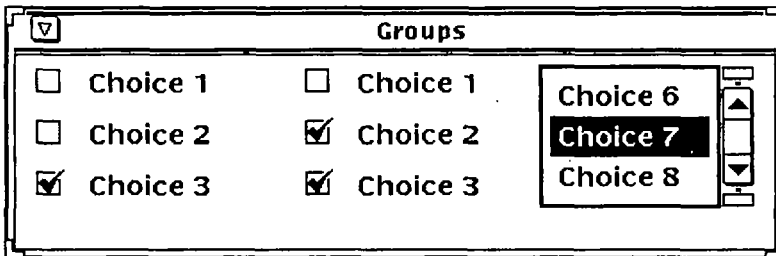
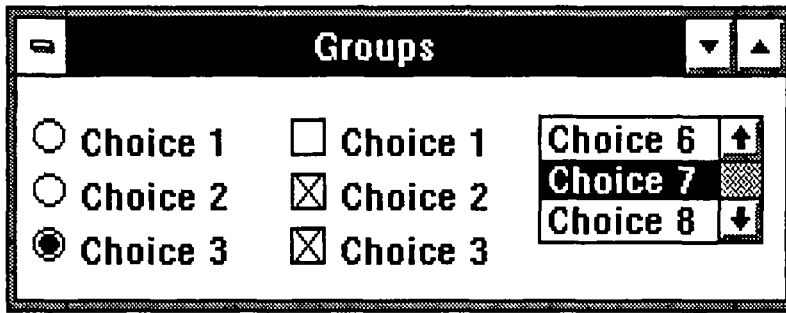
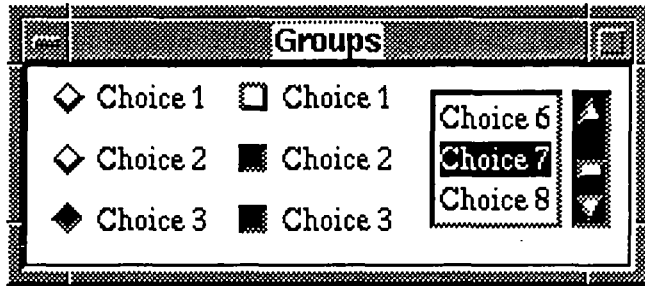
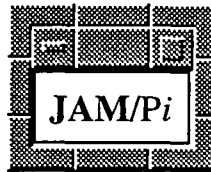


Figure 11: A radio button, checklist, and list box in P//Motif, P//Windows and P//OPEN LOOK.



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.

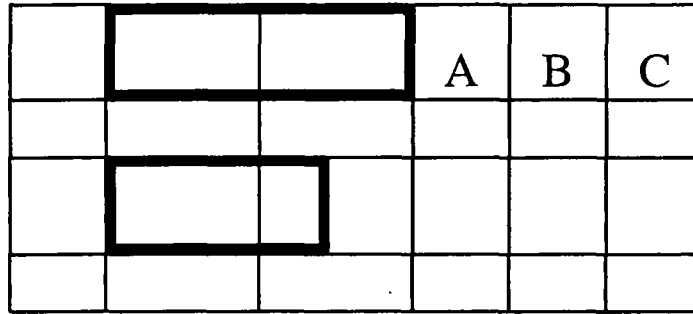


Figure 12: A schematic illustrating the elastic grid. The second and third columns have stretched, as have the first and third rows.

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.

Employee

Employee Information Screen

Name _____ ID# _____

Address _____ SSN - -

City _____ Salary _____

State - Zip - Exemptions _____

Figure 13: A JAM screen in draw mode of Pi/Motif.

Employee

Employee Information Screen

Name [] ID# []

Address [] SSN [- -]

City [] Salary []

State [] Zip [-] Exemptions []

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/Pi**. A set of left justified data entry fields that start in the same column in **JAM** will maintain their left alignment in **JAM/Pi**, 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.

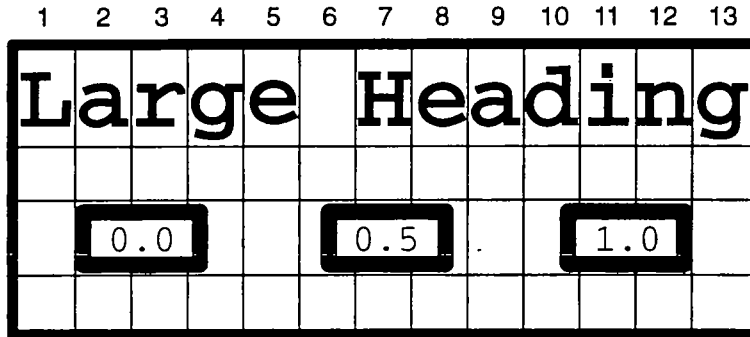


Figure 15: A screen containing a large heading and three data entry fields of length 3. The fields start in columns 2, 6 and 10, respectively. The `halign` of each field is shown as the field's contents.

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.

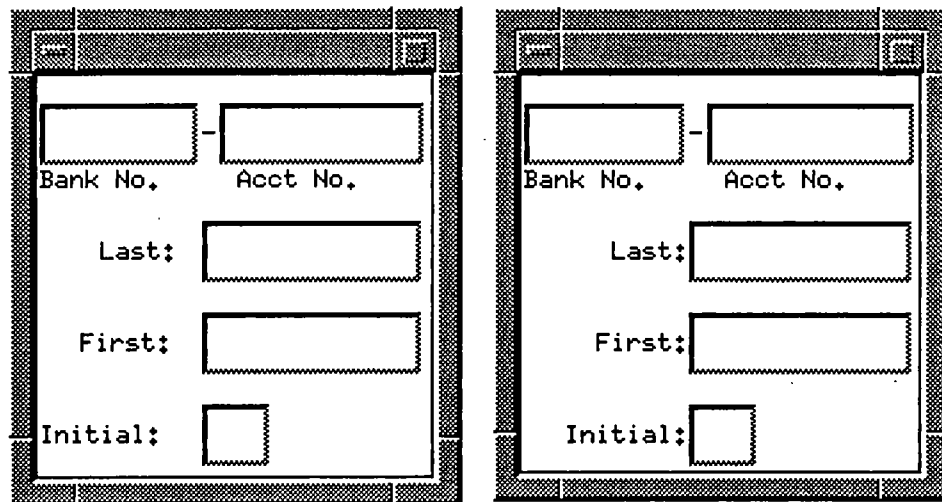


Figure 16: The screen on the left uses display text for the First, Last and Initial labels. The screen on the right uses right justified, protected fields.

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.

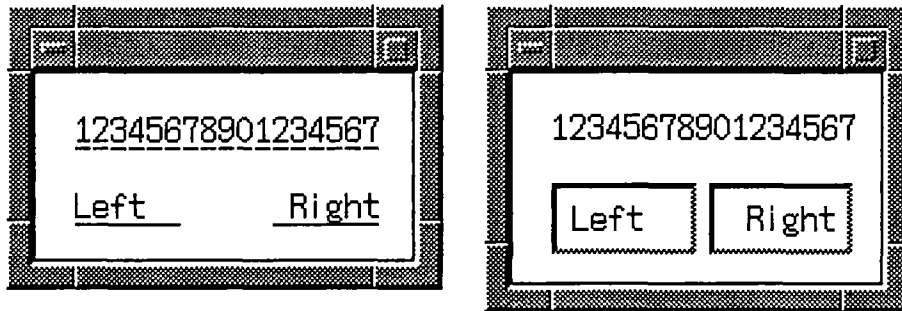


Figure 17: A screen with two fields of length six, shown in draw mode (left) and test mode (right). Left justified widgets expand into whitespace on their right. Right justified widgets expand into whitespace on their left.

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.

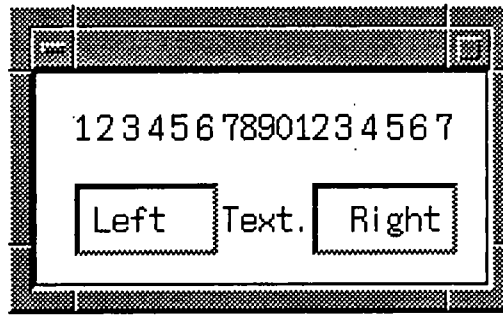


Figure 18: A screen with two fields of length six, and a region of display text. Since there is no room for the widgets to expand into, the grid stretches to accommodate them.

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.

The figure displays two versions of a JAM screen titled "EMPLOYEE TIME OFF". The top version is labeled "Fixed Width" and the bottom version is labeled "Proportional". Both screens contain the following elements: a title bar with the title "EMPLOYEE TIME OFF", a "Name:" label followed by a text input field, a "Days Available:" label, and three checkboxes labeled "sick", "personal", and "vacation". The "Fixed Width" version uses a monospaced font where each character occupies the same horizontal space, resulting in a wider layout. The "Proportional" version uses a font where the width of each character varies according to its shape, resulting in a more compact and visually balanced layout.

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 `Pi/Motifand-Pi/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.

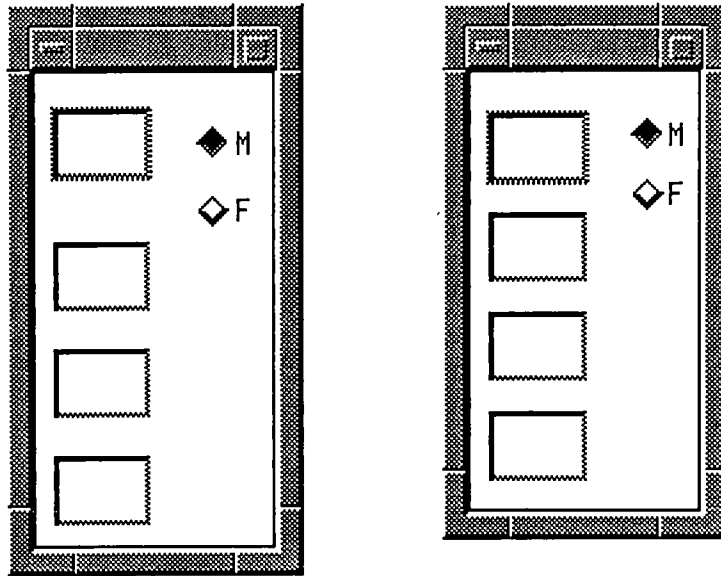


Figure 20: Two screens with a four element array and a radio button. The array is double spaced. The second group item in the radio button falls in the first blank row of the array. Its widget stretches this row. In the left hand screen, the result is an unequally spaced array. The array in the right 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

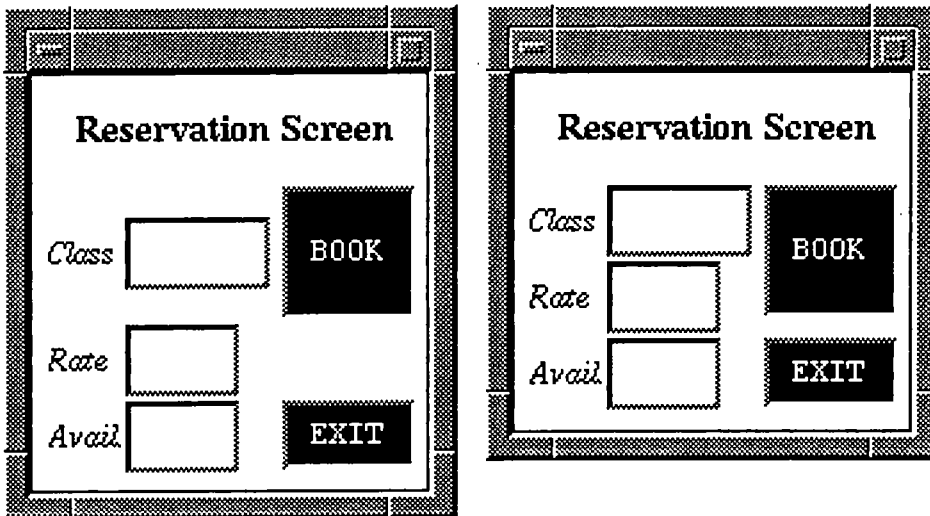


Figure 21: A screen where the `noadj(rows)` field extension is used to prevent a large button from stretching its row.

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.

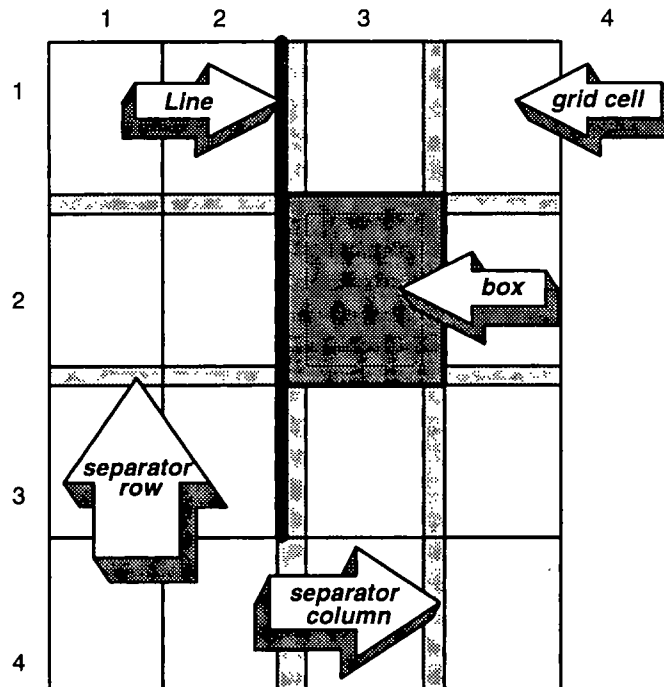


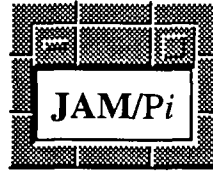
Figure 22: Screen containing two lines and a box. Lines and boxes are drawn in separator rows and columns that are just wide enough to contain the objects and their margins.

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/Pi** behaves, and describes some of the screen level features available in **JAM/Pi**.

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.



In **Pi/Motif** and **Pi/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 Pi/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/Pi under the Windows MDI.

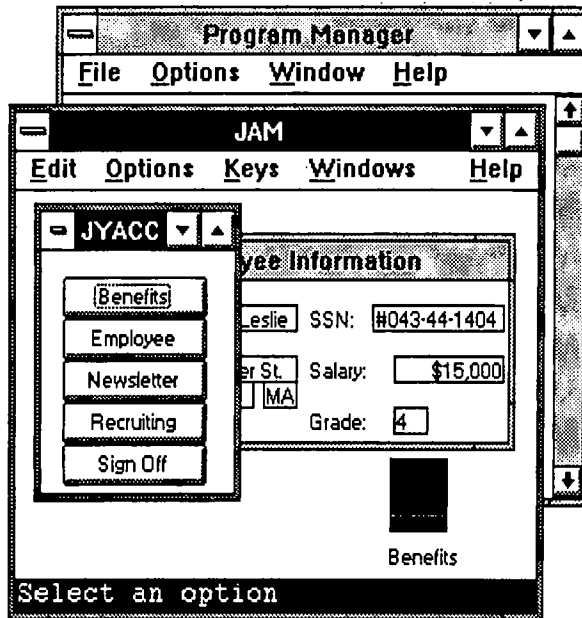


Figure 23: Pi/Windows runs in an MDI frame with a single menu bar at the top and a single status line at the bottom. JAM screens are constrained to move only within the frame.

M

O

In Pi/Motif and Pi/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 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:

W

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.

M

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.

O

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 `nomimize` screen extension removes the minimize button and the minimize entry from the GUI window menu.

4.1.6

Toggleing 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.

<i>Mode in JPL</i>	<i>Equivalent C Function</i>	<i>Message Location</i>
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.

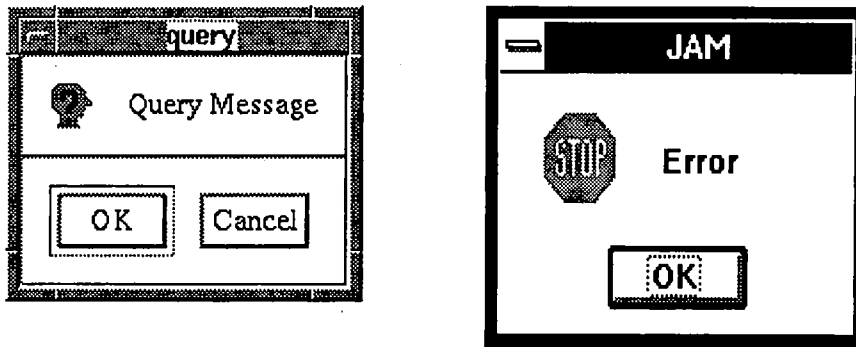


Figure 24: A Motif OK/Cancel dialog (left) and a Windows OK dialog (right).

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.

<i>Character</i>	<i>Meaning</i>	<i>Motif Icon</i>	<i>Windows Icon</i>
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.



In Pi/Windows, `qui_msg` and `quiet` message dialogs contain a stop sign by default if there is no %T in the message text.



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.



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 *Pi/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 *Pi/Motif* and *Pi/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/Pi**, 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/Pi**, 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.

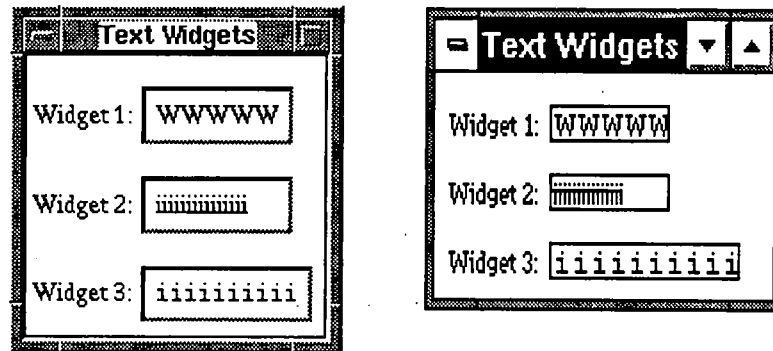


Figure 25: Text widgets in P/Motif and P/Windows.

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/Pi, 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 Pi/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/OPENLOOK, 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.



NOTE: Soft keys are not currently implemented in Pi/Windows.

4.5.1

Location of Soft Keys



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_APPLTC` 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

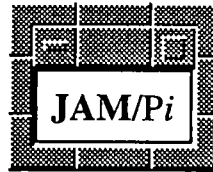
Soft Keys vs. Menu Bars

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 `jkmain.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/Pi**. 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/Pi**. 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.

Form-level GUI Extensions

title?

title:

icon

font

foreground

background

pointer

DECORATIONS:

- ☐ noborder
- ☐ noclose
- ☐ dialog
- ☐ iconify
- ☐ maximize
- ☐ nomaximize
- ☐ nomenu
- ☐ nomimize
- ☐ nomove
- ☐ noresize
- ☐ notitle

BOXES/LINES:

	start		end		
type	row	col	row	col	
> <input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

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.

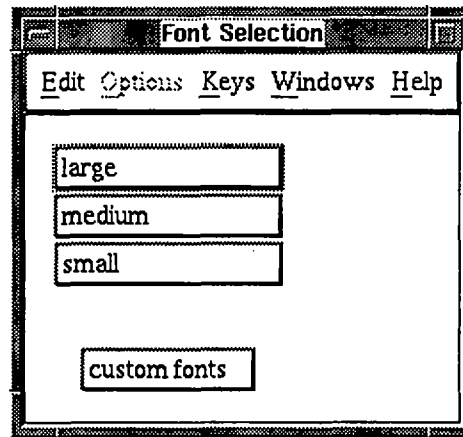


Figure 27: An item selection screen with a list of user-defined font aliases.

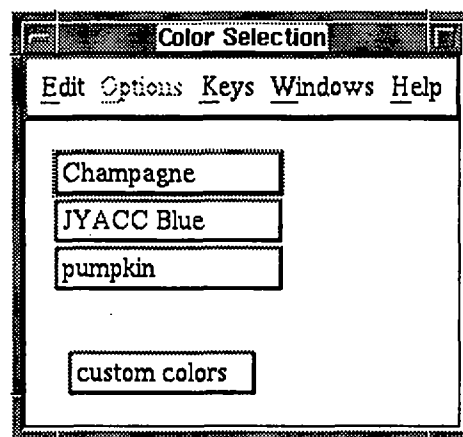


Figure 28: An item selection screen with a list of user-defined color aliases.

- 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 Pi/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

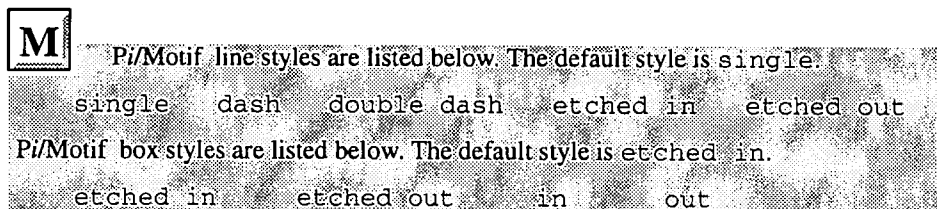
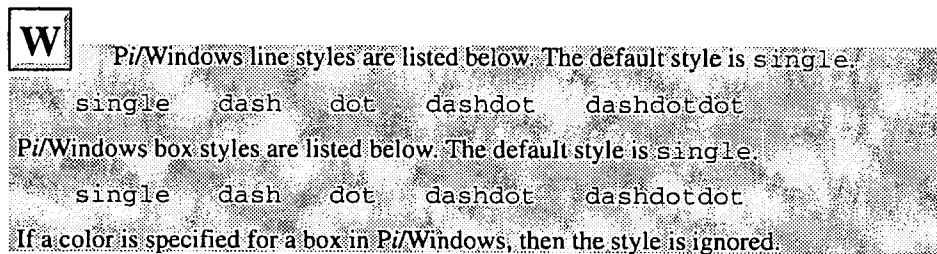
Specify the appearance of a line or box by popping up the details window, described below. A sample details window is shown in Figure 29. The items on this screen provide the arguments to the hline, vline, and box screen extensions.

● 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.



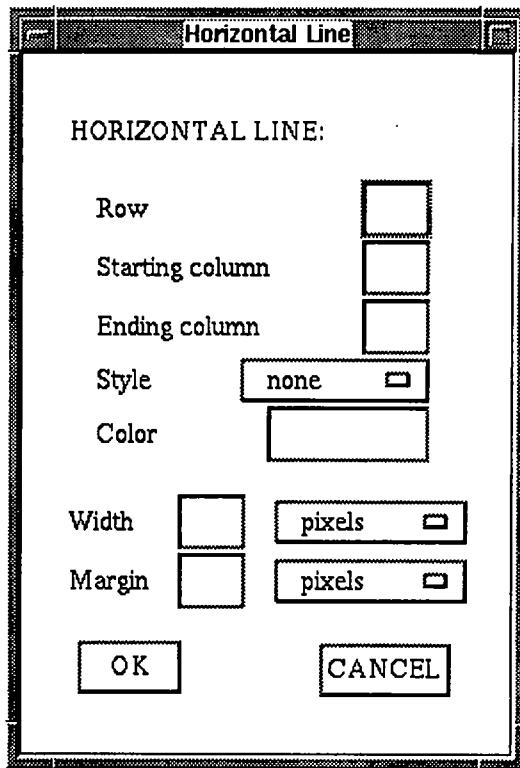


Figure 29: The details window for a horizontal line. There are similar windows for vertical lines and boxes.

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.

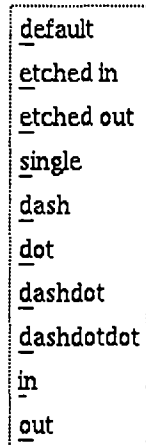


Figure 30: The styles option menu.

- **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:

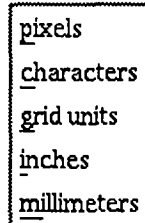


Figure 31: The units option menu.

- **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/Pi 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/Pi** 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/Pi** asks you whether you want to adjust each relevant edit upon transmitting out of the screen. If this option is not selected, **JAM/Pi** makes the adjustments without consulting you.

5.3.2

Forcing the Widget Type

If the option "force widget type" is selected, **JAM/Pi** 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.

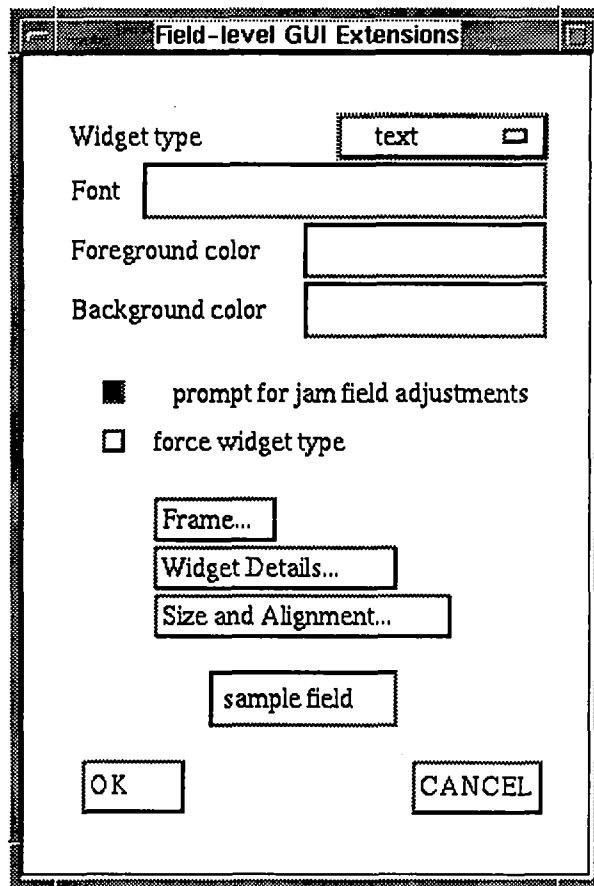


Figure 32: The Field Extensions window.

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:

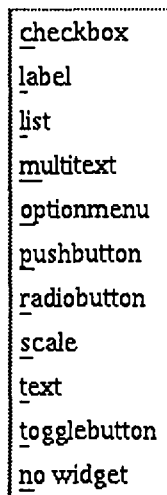


Figure 33: The widget type option menu.

- 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/Pi** 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/Pi** 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/Pi** 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

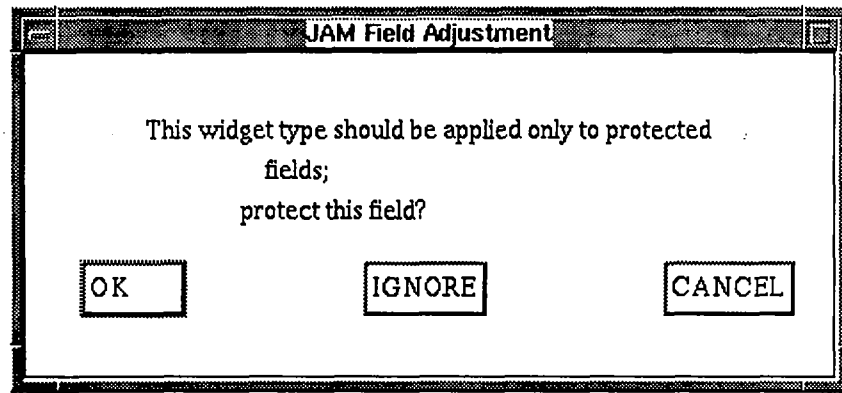


Figure 34: A sample field adjustment dialog box.

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/Pi** 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.



Pi/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 Pi/Windows, then the style is ignored.



Pi/Motif frame styles are listed below. The default style is etched in.

etched in etched out in out

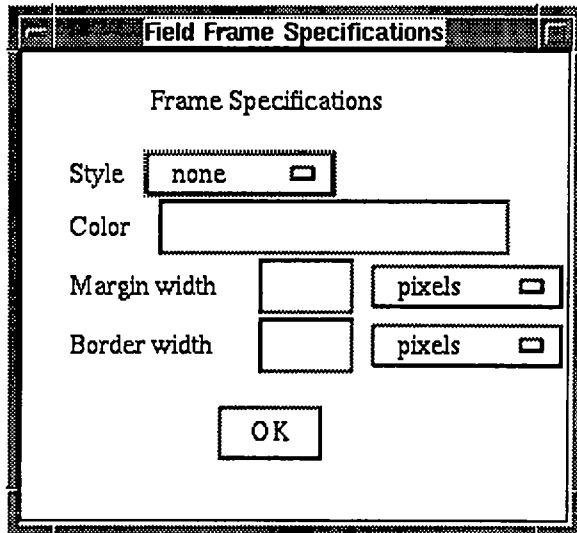


Figure 35: The field frame specifications window.



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.



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.

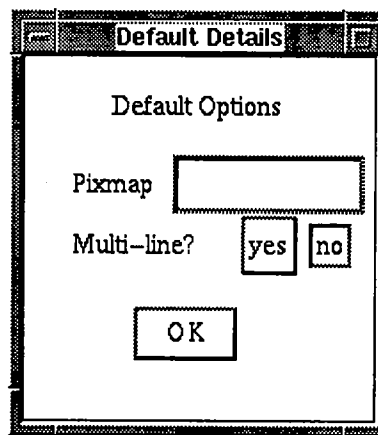


Figure 36: The widget details screen for: checklists, labels, push buttons, radio buttons and toggle buttons.

- **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.

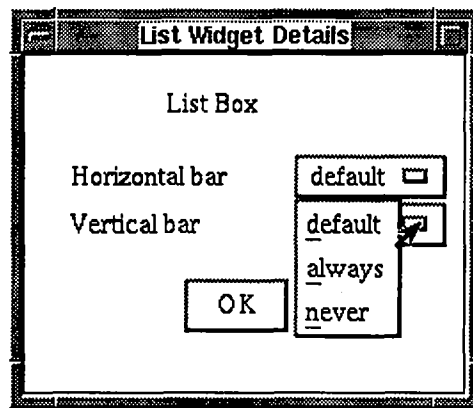


Figure 37: The widget details screen for list boxes and multiline text widgets. Notice that the option menu for vertical bar is posted.

■ 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.

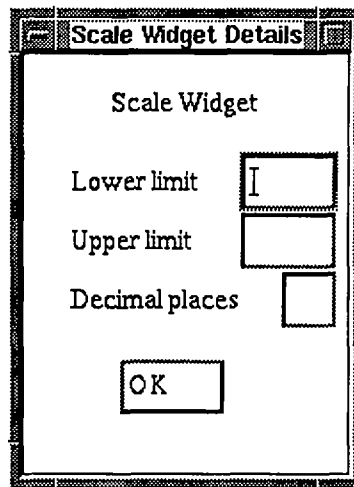


Figure 38: The details screen for a scale widget.

- 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).

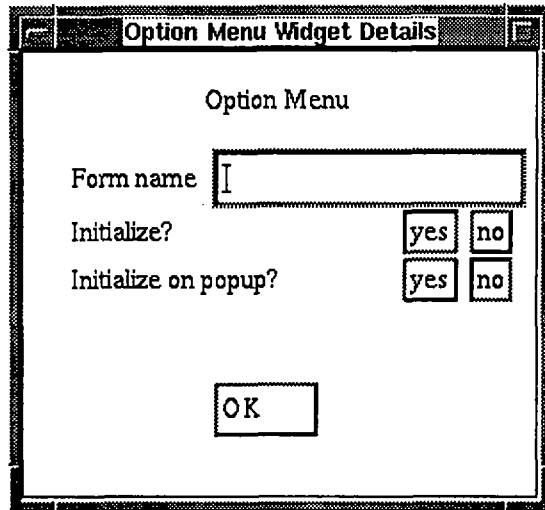


Figure 39: The option menu widget detail screen.

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/Pi 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.

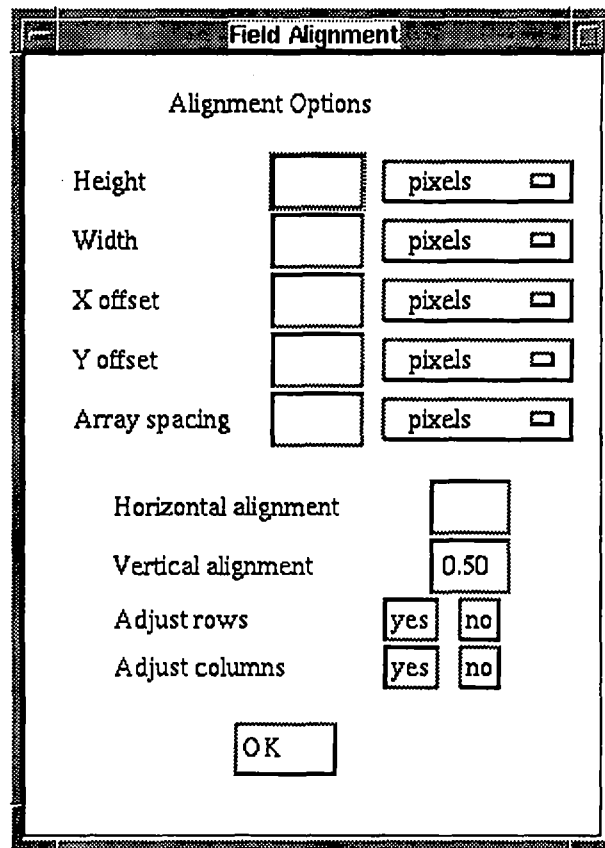


Figure 40: Screen for entering field size and alignment options.

- 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 its 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.

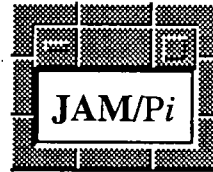
- Adjust rows (`noadj`)

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/Pi.

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 Pi/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.



In `Pi/Motif` and `Pi/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		
<i>Type</i>	<i>Syntax</i>	<i>Page</i>
Incremental Positioning		
Height	height(<i>value</i> [<i>units</i>])	96
Width	width(<i>value</i> [<i>units</i>])	96
Horizontal Offset	hoff(<i>distance</i> [<i>units</i>])	102
Vertical Offset	voff(<i>distance</i> [<i>units</i>])	102
Horizontal Alignment	halign(<i>value</i>)	94
Vertical Alignment	valign(<i>value</i>)	94
Disable Adjustment	noadj(<i>direction</i>)	115
Equally Space an Array	space(<i>distance</i> [<i>units</i>])	140
Fonts, Colors and Decorations		
Foreground Color	fg(<i>color</i>)	81
Background Color	bg(<i>color</i>)	81
Font	font(<i>fontname</i>)	89
Bitmapped Image	pixmap(<i>name</i>)	130
Frame	frame ([<i>style</i> , <i>color</i> , <i>matte</i> , <i>margin</i>])	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 [(<i>selectscreen</i> , <i>init</i> , <i>popup</i>)]	127
Multiline Text Widget	multitext [(no hbar, no vbar)]	113

<i>Field Extensions</i>		
<i>Type</i>	<i>Syntax</i>	<i>Page</i>
Multiline Button	multiline	111
No Widget	nowidget	126
Push Button	pushbutton	136
Radio Toggle Button	radiobutton	138
Scale Widget	scale (<i>min</i> , <i>max</i>)	139
Text Widget	text	141

Screen Extensions		
<i>Type</i>	<i>Syntax</i>	<i>Page</i>
Fonts and Colors		
Font	font(<i>fontname</i>)	89
Foreground Color	fg(<i>color</i>)	81
Background Color	bg(<i>color</i>)	81
Lines and Boxes		
Horizontal Line	hline(<i>r</i> , <i>c1</i> , <i>c2</i> [, <i>style</i> , <i>color</i> , <i>width</i> , <i>margin</i>])	98
Vertical Line	vline(<i>c</i> , <i>r1</i> , <i>r2</i> [, <i>style</i> , <i>color</i> , <i>width</i> , <i>margin</i>])	98
Box	box(<i>l1</i> , <i>c1</i> , <i>l2</i> , <i>c2</i> [, <i>style</i> , <i>color</i> , <i>matte</i> , <i>margin</i>])	84
Screen Behavior		
Associate Icon with Screen and Allow Iconification	icon(<i>name</i>)	104
Start the Screen as an Icon	iconify	106
Specify the Pointer Shape	pointer(<i>cursor</i>)	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(<i>string</i>)	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.



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.



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.



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



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.



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.



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:



single dash dot dashdot dashdotdot

single is the default box style in Pi/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.



etched in etched out in out

etched in is the default box style in Pi/Motif



single

Style is ignored in Pi/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 Pi/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/Pi 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 Pi/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 Pi/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.

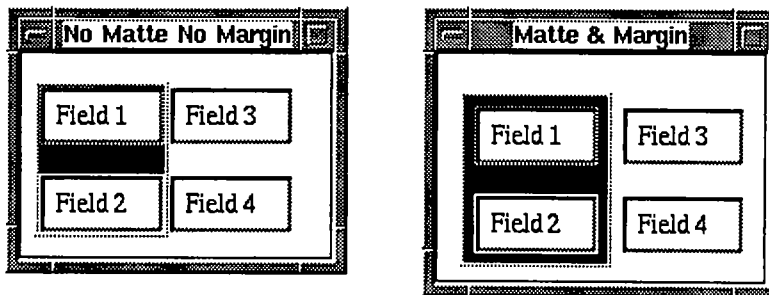


Figure 41: Two screens with black boxes. The box around fields 1 and 2 on the left hand screen has no matte or margins. The box on the right hand screen has a 5 pixel matte and a 5 pixel margin.

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.

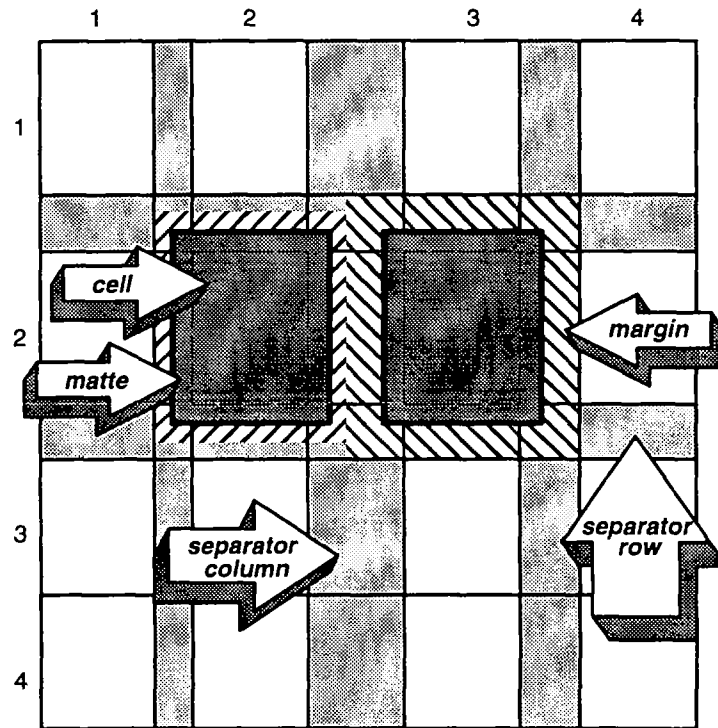


Figure 42: Two one-cell boxes that have different margins. The edges of the boxes are drawn in separator rows and columns that are just wide enough to accommodate the matte, lines and margins.

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.



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.

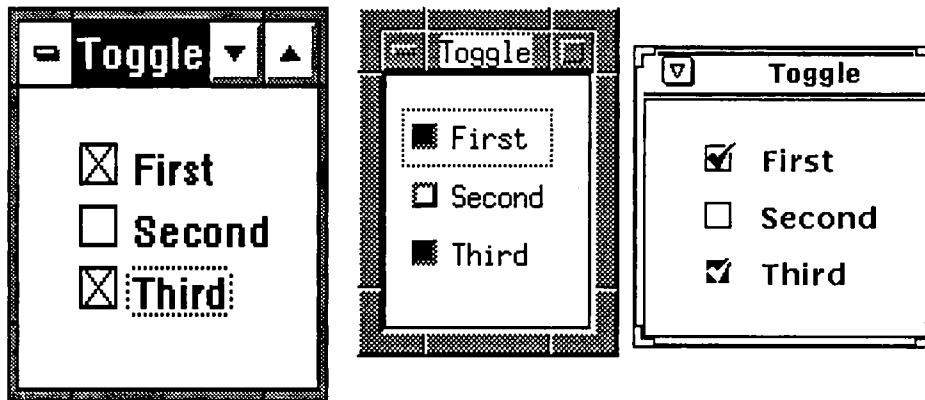


Figure 43: Checklist style toggle buttons in Windows, Motif and OPEN LOOK.

<<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.

W

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.

M

O

NOTE: This extension is not supported in Pi/Motif or Pi/OPEN LOOK.

<>

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*~~ ~~*r*~~ ~~*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 <> 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 Pi/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 Pi/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.

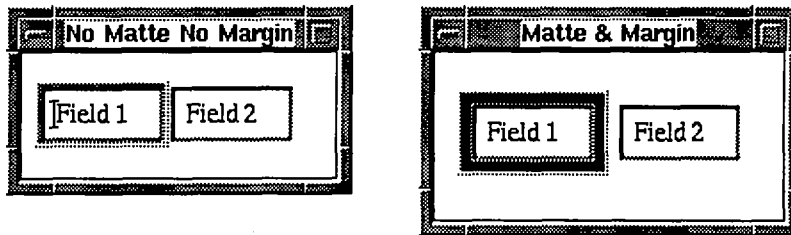


Figure 44: Two screens with a framed field. The frame around field 1 on the left hand screen has no matte or margin. The frame on the right hand screen has a 5 pixel matte and a 5 pixel margin.

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:

<i>Suffix</i>	<i>Units</i>	<i>Description</i>
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.)

<i>Suffix</i>	<i>Units</i>	<i>Description</i>
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.



In Pi/Windows, bitmaps are scaled to fit in the height and width specified.



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.

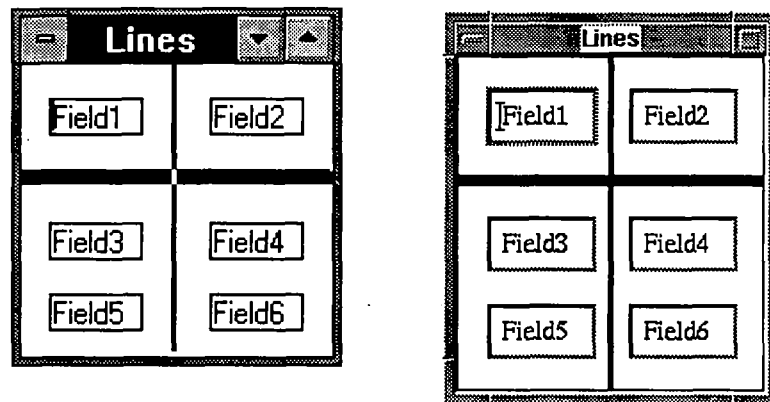


Figure 45: Screens broken into quadrants by horizontal and vertical lines.

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:



single dash dot dashdot dashdotdot



single dash double dash etched in etched out



single dash

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.



In Pi/Windows, if no color is specified, then JAM/Pi selects a color that is visible against the background.



In Pi/Motif, line coloring is style dependent.

For the 3-D line styles (etched in and etched out) JAM/Pi ignores the color specification, and uses colors that show up against the background.

For the other line styles, the specified color is used. If no color is specified, then the background color of the form is used. This means that the line is not visible against the background.



If no color is specified, then the background color of the form is used. This means that the line is not visible against the background.

width specifies the width of a line, provided that the style is **single**.



In Pi/Windows, if the style is not **single**, the width is ignored.



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.

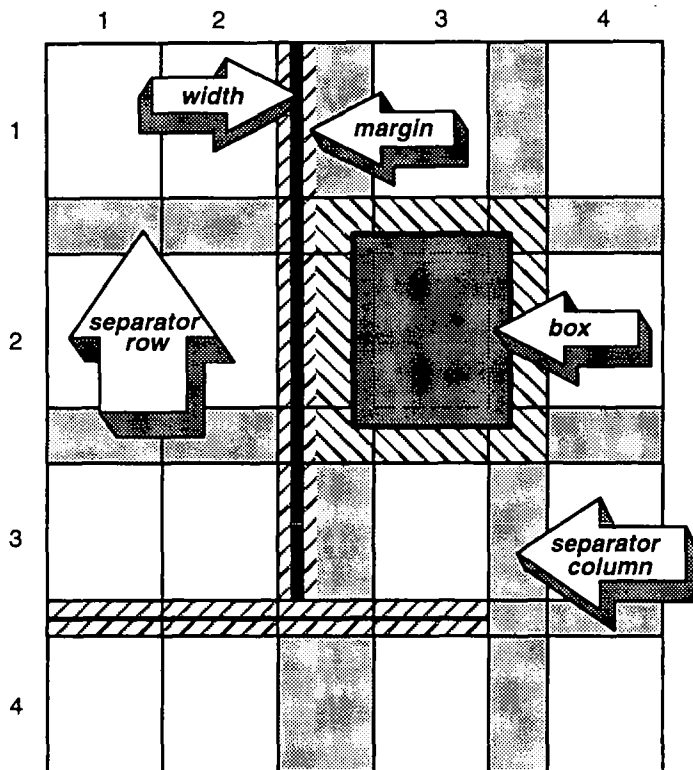


Figure 46: Screen containing two lines and a box. The vertical line is specified for column 3, the horizontal line for column 4. Lines and boxes are drawn in separator rows and columns that are sized just wide enough for them.

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.

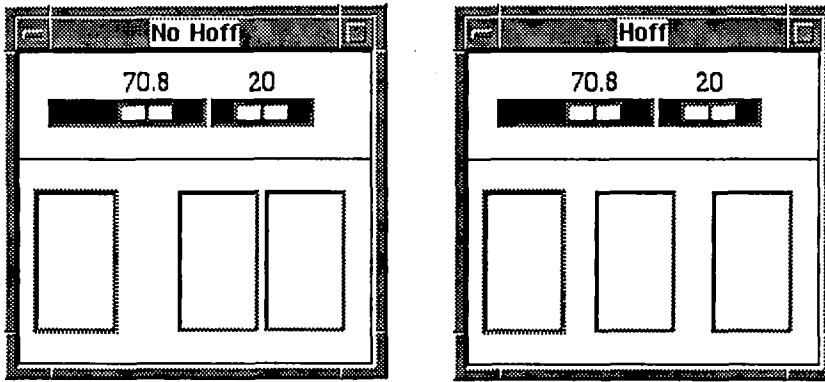


Figure 47: Screens with two scale widgets and three multiline text widgets. In the left hand screen, an oversized scale widget at top left has stretched the grid, causing unequal spacing between the widgets below it. In the right hand screen, an `hoff` screen extension on the middle multiline widget 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/Pi**.

W

In **Pi/Windows** display text does not become a label widget. Instead, it is simply text painted on the screen.

Figure 48 illustrates label widgets.

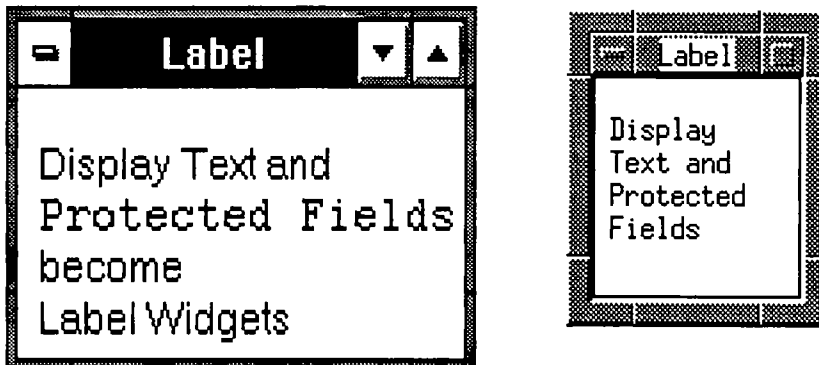


Figure 48: Label widgets in Windows and Motif.

<<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.

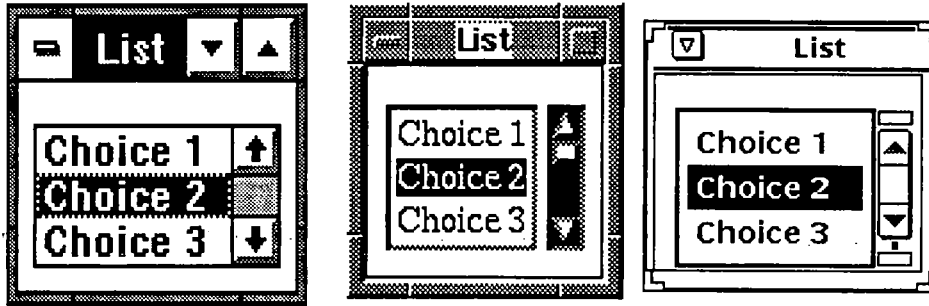


Figure 49: List boxes with vertical scroll bars in Windows, Motif and OPEN LOOK.

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.



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.



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.



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 Button with 3 lines

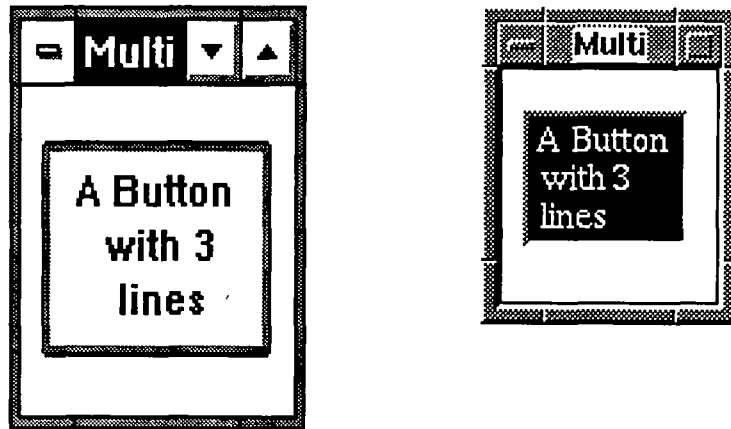


Figure 50: A multiline button in Windows (left) and Motif (right).

<<multitext>>

create a multiline text widget from an array

SYNOPSIS

```
# <<multitext [(no hbar, no vbar)] >>
```

TYPE

Field Extension

DESCRIPTION

An array in **JAM/Pi** 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.

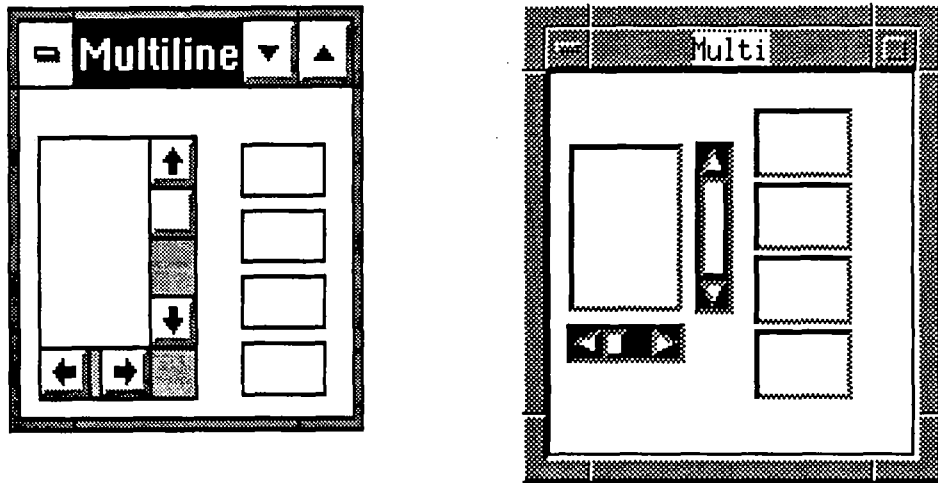


Figure 51: Multiline text widgets versus regular arrays in Windows and Motif.

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/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 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.



In *Pi/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.



In *Pi/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.

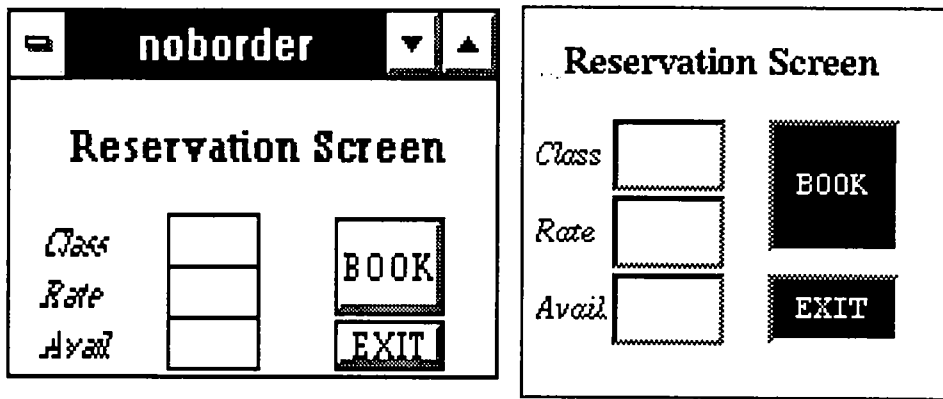


Figure 52: noborder screens in P/Windows and P/Motif.

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 Pi/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.

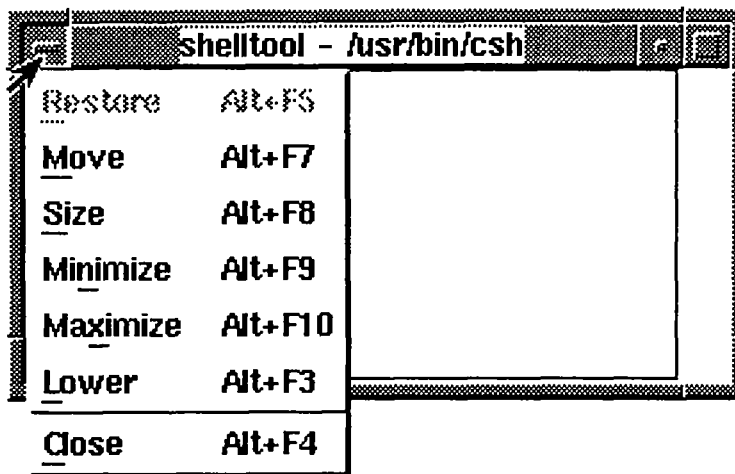


Figure 53: The GUI window menu in Motif.

<<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.



In Pi/Motif, only screens that have the `icon` screen extension may be minimized by the user. The `nominimize` extension is therefore rarely used.



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.

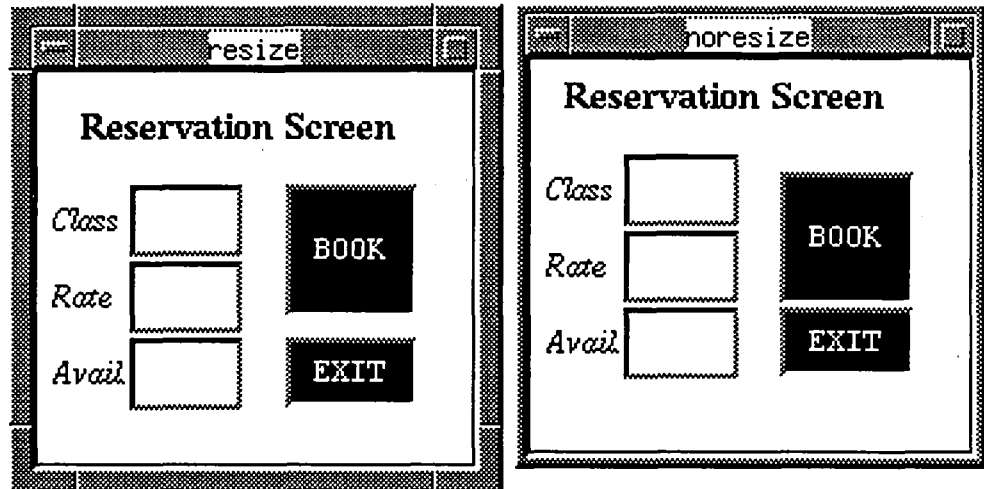


Figure 54: A Motif screen with and without resize handles.

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.

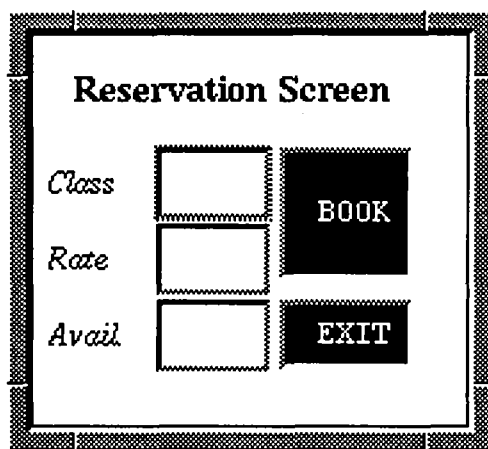


Figure 55: A screen with the `notitle` extension. It has no title bar, minimize button, maximize button or GUI window menu button.

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.



In Pi/Motif and Pi/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.

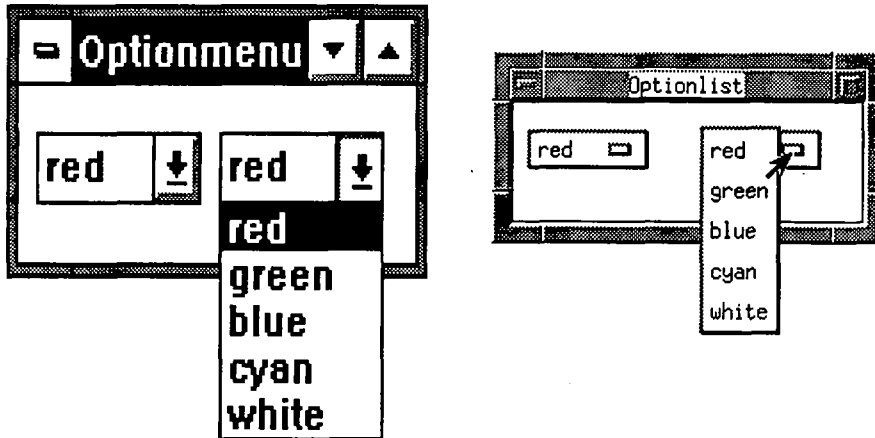


Figure 56: Option menus in Windows (left) and Motif (right). The right hand widget on each screen has its pop-up posted.

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.

W

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.

M

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 *pixmap*s replaces the word *bitmap*s (eg. - \$HOME/*pixmap*s).

If neither a bitmap nor a pixmap is found, the default bitmap is used.

O Under OPEN LOOK, 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
$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/Pi 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/Pi searches for a pixmap. The paths are the same as above, except that XPMLANGPATH replaces XBMLANGPATH and the word *pixmap*s replaces the word *bitmap*s (eg. - \$HOME/pixmap).

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 Pi/Motif, pointer shapes are listed in the file `/usr/include/X11/cursorfont.h`.

O

In Pi/OPEN LOOK, pointer shapes are listed in the file `/$OPENWINHOME/include/X11/cursorfont.h`.

W

In Pi/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.



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 `Pi/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.

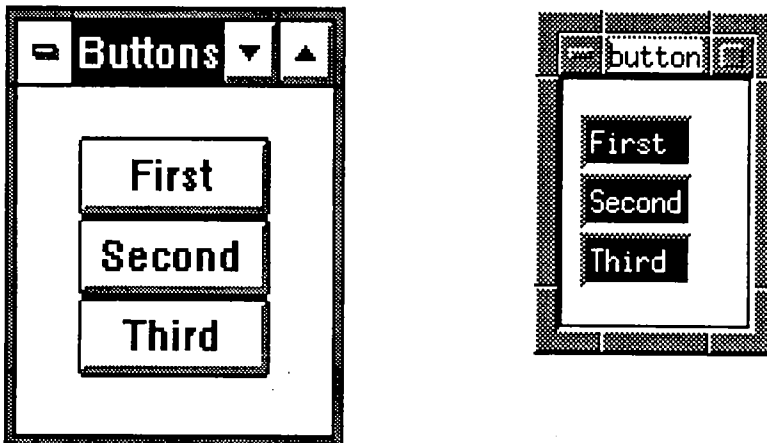


Figure 57: Push Buttons in Windows and Motif

<<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.



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.

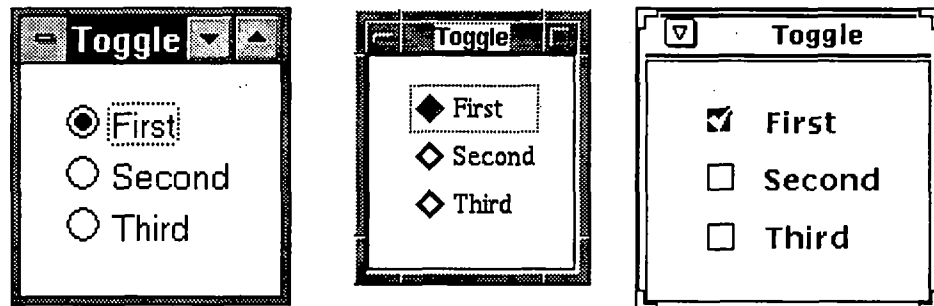


Figure 58: Radio buttons in Windows, Motif and OPEN LOOK.

<<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.

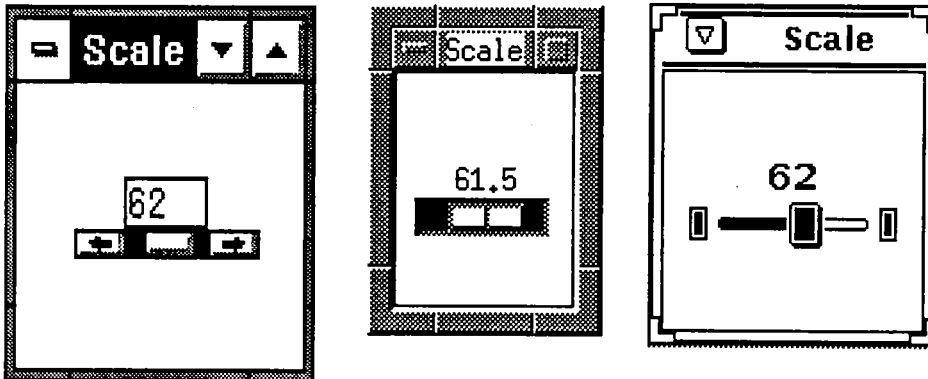


Figure 59: Scale widgets in Windows, Motif and OPEN LOOK.

<<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.

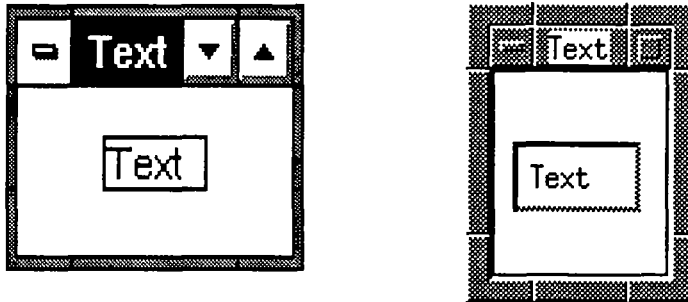


Figure 60: A Text widget in Windows and in Motif.

<<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 *Pi/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 *Pi/Motif* and *Pi/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.

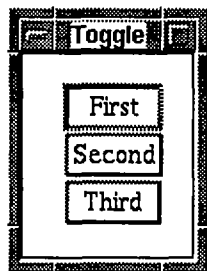
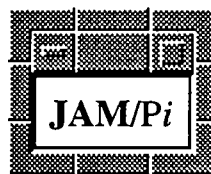


Figure 61: A set of Motif in/out style toggle buttons. The first item is selected.



Chapter 7

Setting Application Defaults

Each GUI provides its own method for setting defaults. *Pi/Motif* uses *resource files*, and *Pi/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/Pi** 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 *Pi/Motif* and *Pi/OPEN LOOK* is named `myapp`, and the application specific initialization file in *Pi/Windows* is named `myapp.ini`. The default value for this argument in the distributed software is "XJam" in *Pi/Motif*, "OLJam" in *Pi/OPEN LOOK* and "Jam" in *Pi/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.



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_X11init` (see above). The class name specified in `sm_X11init` 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_X11init` (see above). The class name specified in `sm_X11init` 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/Pi offers access to many more color choices than character JAM. Resource and initialization files provide a mapping between JAM colors and GUI colors. JAM/Pi 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 un-highlighted. 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/Pi 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

W

In Pi/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.

M

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
```

O

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/Pi 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.

W

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:

<code>SYSTEM_FONT</code>	a proportional font (Windows uses it in pull-down menus).
<code>SYSTEM_FIXED_FONT</code>	a fixed width font. This is the font used in Draw Mode to complement <code>SYSTEM_FONT</code> .
<code>OEM_FIXED_FONT</code>	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.
<code>ANSI_FIXED_FONT</code>	Courier, fixed width.
<code>ANSI_VAR_FONT</code>	Helvetica, proportional.

Other Windows font specifications cannot be used with the `SystemFont` setting.

M

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.

O

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/Pi 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

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. 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 naming¹

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*.

Copyright © 1988 by the Massachusetts Institute of Technology.

Copyright © 1988, 1989 by Digital Equipment Corporation. All rights reserved.

Permission to use, copy, modify, and distribute this appendix for any purpose and without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies. MIT and Digital Equipment Corporation make no representations about the suitability for any purpose of the information in this document. This document is provided "as is" without express or implied warranty.

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*
```


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/Pi 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 *Pi/Motif* and *Pi/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/Pi* palette.

W

In *Pi/Windows*, a font alias *may not* be used for the `SystemFont` setting in the initialization file.



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 Pi/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 Pi/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 *Pi/Motif* and *Pi/OPEN LOOK*.

7.6.1

Motif and OPEN LOOK Behavioral Resources

Three resources control the behavior of *JAM/Pi* 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/Pi*. 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.

O 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.

O 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.

O 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	deep sky blue	light sky blue	papaya whip
antique white	dim gray	light slate blue	peach puff
aquamarine	dim grey	light slate gray	peru
azure	dodger blue	light slate grey	pink
beige	firebrick	light steel blue	plum
bisque	floral white	light yellow	powder blue
black	forest green	lime green	purple
blanched almond	gainsboro	linen	red
blue	ghost white	magenta	rosy brown
blue violet	gold	maroon	royal blue
brown	goldenrod	medium blue	saddle brown
burlywood	gray	medium orchid	salmon
cadet blue	green	medium purple	sandy brown
chartreuse	green yellow	medium sea green	sea green
chocolate	grey	medium slate blue	sienna
coral	honeydew	medium turquoise	sky blue
cornflower blue	hot pink	medium violet red	slate blue
cornsilk	indian red	midnight blue	slate gray
cyan	ivory	mint cream	slate grey
dark goldenrod	khaki	misty rose	snow
dark green	lavender	moccasin	spring green
dark khaki	lavender blush	navajo white	steel blue
dark olive green	lawn green	navy	tan
dark orange	lemon chiffon	navy blue	thistle
dark orchid	light blue	old lace	tomato
dark salmon	light coral	olive drab	turquoise
dark sea green	light cyan	orange	violet
dark slate blue	light goldenrod	orange red	violet red
dark slate gray	light gray	orchid	wheat
dark slate grey	light grey	pale goldenrod	white
dark turquoise	light pink	pale green	white smoke
dark violet	light salmon	pale turquoise	yellow
deep pink	light sea green	pale violet red	yellow green

7.7

MOTIF RESOURCE OPTIONS

The following sections describe resources and options that are particular to Pi/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.

<i>Resource</i>	<i>Type</i>	<i>Command Line</i>	<i>Description</i>
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.

<i>Resource</i>	<i>Type</i>	<i>Command Line</i>	<i>Description</i>
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 Pi/Motif:

```
xjxform -fn '-*courier*r*12' myscreen.jam
```

7.7.2

Widget Hierarchy in Pi/Motif

Widgets are arranged in a parent-child hierarchy. The following tables describe the widget hierarchy in Pi/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:

<i>Widget Class</i>	<i>Name</i>
ApplicationShell... (class given by sm_X11init)	<i>application-name</i>
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:

<i>Widget Class</i>	<i>Name</i>
...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:

<i>Object</i>	<i>Widget Class</i>	<i>Name</i>
Data Entry Field	...XmText	<i>field-name</i>
Data Entry Field with Indicators	...XmDrawingArea	<i>field-name</i>
	XmText*	<i>field-name</i>
	XmArrowButton	indicator
	:	:
	XmArrowButton	indicator
Protected Field	...XmLabel	<i>field-name</i>
Menu Field	...XmPushButton	<i>field-name</i>
Group Member	...XmToggleButton	<i>field-name</i>
Multiline Text	...XmText	<i>field-name</i>

<i>Object</i>	<i>Widget Class</i>	<i>Name</i>
Multiline Text with Scrollbars	...XmScrolledText	<i>field-nameSW</i>
	XmText*	<i>field-name</i>
List Box	...XmList	<i>field-name</i>
List Box with Scroll Bars	...XmScrolledList	<i>field-nameSW</i>
	XmList*	<i>field-name</i>
Optionmenu	...XmRowColumn*	<i>field-name</i>
	...XmMenuShell	popup_ <i>field-name</i> _pane
	XmRowColumn	<i>field-name</i> _pane
	XmPushButton	<i>label-text</i>
	:	:
	XmPushButton	<i>label-text</i>
Scale	...XmScale	<i>field-name</i>
	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:

<i>Object</i>	<i>Widget Class</i>	<i>Name</i>
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:

<i>Object</i>	<i>Widget Class</i>	<i>Name</i>
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:

<i>Menu Script Keyword</i>	<i>Widget Class</i>	<i>Name</i>
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:

<i>Object</i>	<i>Widgets Class</i>	<i>Name</i>
Windows Menu	...XmRowColumn	windows
	XmPushButton	window-name
	:	:
	XmPushButton	window-name
	XmSeparator	sep1
	XmPushButton	windows_raise

<i>Object</i>	<i>Widgets Class</i>	<i>Name</i>
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 Pi/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.

<i>Resource</i>	<i>Type</i>	<i>Command Line</i>	<i>Description</i>
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 .

<i>Resource</i>	<i>Type</i>	<i>Command Line</i>	<i>Description</i>
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 Pi/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 re-size 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 Pi/OPEN LOOK

Widgets are arranged in a parent-child hierarchy. The following tables describe the widget hierarchy in Pi/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:

<i>Widget Class</i>	<i>Name</i>
<code>ApplicationShell...</code> (class given by <code>sm_X11init</code>)	<i>application-name</i>
Form	main
Form	workarea
Control	softkeys
OblongButton	softkey
:	:
OblongButton	softkey
StaticText	status
Control	menubar
MenuButton	Edit
MenuButton	Windows
MenuButton	<i>menu-name</i>
:	:
MenuButton	<i>menu-name</i>

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:

<i>Widget Class</i>	<i>Name</i>
...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:

<i>Object</i>	<i>Widget Class</i>	<i>Name</i>
Data Entry Field	...TextField	<i>field-name</i>
Protected Field	...StaticText	<i>field-name</i>
Menu Field	...OblongButton	<i>field-name</i>
Checklist	...CheckBox	<i>field-name</i>
Radio Button	...RectButton	<i>field-name</i>
Multiline Text	...TextEdit	<i>field-name</i>
List Box	...ScrollingList	<i>field-name</i>
Optionmenu	...Control	<i>field-nameC</i>
	StaticText*	<i>field-name</i>
	AbbrevMenuButton	<i>field-nameB</i>
	...MenuShell	menu
	Form	menu_form
	Control	pane
	OblongButton	<i>label-text</i>
	:	:
	OblongButton	<i>label-text</i>
Scale	...Control	<i>field-nameC</i>
	StaticText	<i>field-nameT</i>
	Slider*	<i>field-name</i>

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, OLJam*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:

<i>Object</i>	<i>Widget Class</i>	<i>Name</i>
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:

<i>Object</i>	<i>Widget Class</i>	<i>Name</i>
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 menu's form. The hierarchy for items on menus and submenus is identical. It is as follows:

<i>Menu Script Keyword</i>	<i>Widget Class</i>	<i>Name</i>
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:

<i>Object</i>	<i>Widgets Class</i>	<i>Name</i>
Windows Menu	...MenuButton	windows
	...MenuShell	menu
	Form	menu-form
	Control	pane
	OblongButton	window-name
	:	:
	OblongButton	window-name
	Stub	sepl
	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.
```

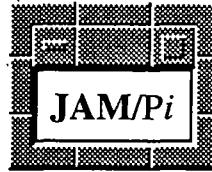
```
!
```

```
! 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/Pi. 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 Pi/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.



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



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.



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.

<i>Scope</i>	<i>Description</i>	<i>Location in Motif/OPEN LOOK</i>
KS_APPLIC	Application-level menu bar.	Base screen or pop-up.
KS_FORM	Screen-level menu bar.	Local to form if <code>formMenus</code> 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 <code>formMenus</code> 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 <code>jxform</code> . 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.

- **menu***name*

identifies the menu. Any **display options** specified directly after the **menu***name* 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 Pi/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 Pi/Motif and Pi/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.

<i>Display Option</i>	<i>Actions</i>	<i>Description</i>
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 <code>indicator</code> option is not also specified, this option shifts the menu. Indicators are ignored on horizontal menu bars.

<i>Display Option</i>	<i>Actions</i>	<i>Description</i>
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.

<i>Display Option</i>	<i>Interface</i>	<i>Description</i>
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

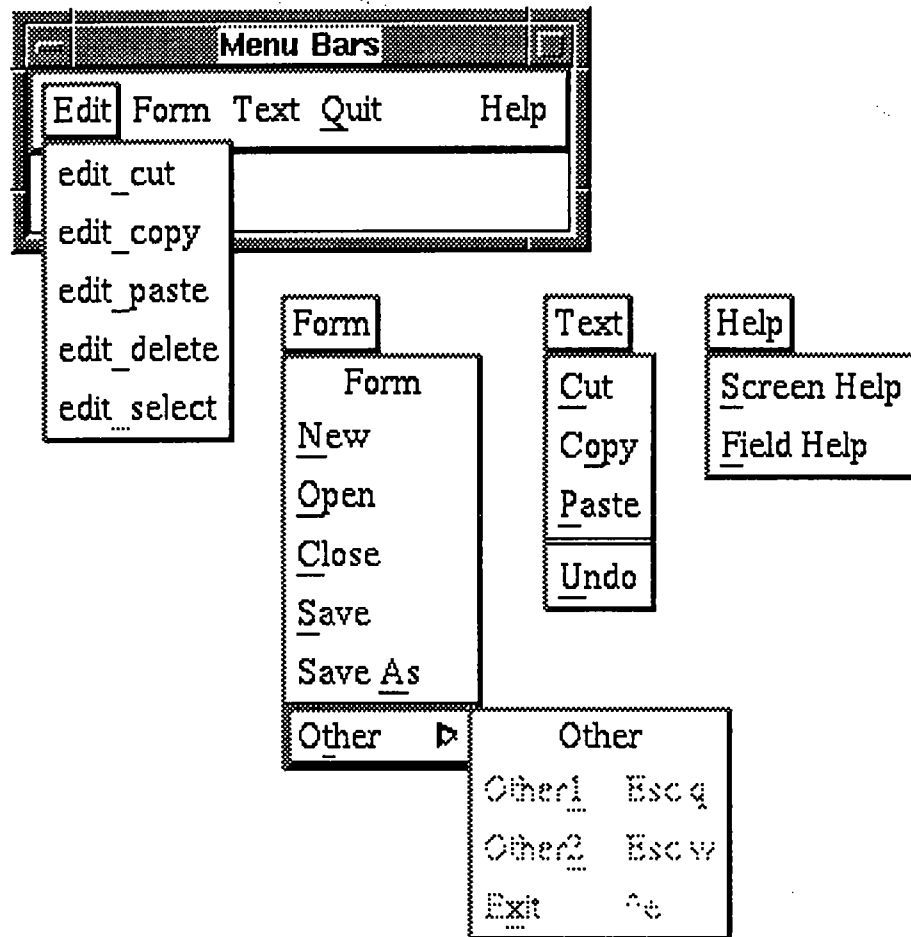


Figure 63: The menu bars produced by the sample menu script.

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.

<i>Routine</i>	<i>Description</i>
<code>sm_c_menu</code>	close a menu bar
<code>sm_d_menu</code>	display a menu bar stored in memory
<code>sm_menuinit</code>	initialize menu bar support
<code>sm_mn_forms</code>	install menu bars in memory (in a custom executive)
<code>sm_mnadd*</code>	add an item to the end of a menu bar
<code>sm_mnchange*</code>	alter a menu bar item (eg- grey out an item)
<code>sm_mndelete</code>	delete a menu bar item
<code>sm_mnget*</code>	get menu bar item information
<code>sm_mninsert*</code>	insert a new menu bar item
<code>sm_mnitems</code>	get the number of items on a menu bar
<code>sm_mnnew</code>	create a new menu bar by name
<code>sm_r_menu</code>	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

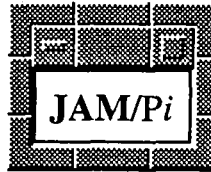
Converting Keysets into Menu Bars

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 bars 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/Pi** is similar to that in **Jterm** 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 **Pi/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 **Pi/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 Pi/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 Pi/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 Pi/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

Focus

W

- In Pi/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

- In Pi/Motif and Pi/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.

O

- 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 Pi/Motif and Pi/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 `unset-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 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.



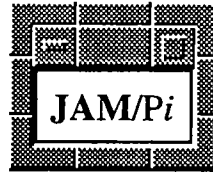
- In Pi/Windows, you can cut or copy text in a text widget, and then paste it into another text widget (or onto the screen as display text in Draw Mode). Drag across text to select it. Choose Cut or Copy.
- In Pi/Motif, you can cut, copy, and paste text in text widgets. Drag across text to highlight 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. 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//MOTIF AND P//OPEN LOOK



Normally Motif and OPEN LOOK do not support overstrike mode. P//Motif and P//OPEN LOOK *do* support overstrike mode in text widgets. In fact, overstrike mode is the default text entry mode in JAM/Pi, just as it is in character JAM.

10.2

INTERFACING WITH THE GUI LIBRARY

JAM/Pi 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/Pi 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 Pi/Windows.

A demonstration program that uses external graphics is provided in source form with JAM/Pi. It is called `winpie` in Pi/Windows, `xpie` in Pi/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 Pi/WINDOWS



In Pi/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

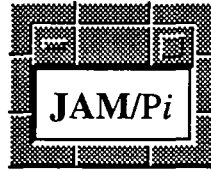
```
!chkdsk.pif
```

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/Pi.

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/Pi 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/Pi 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

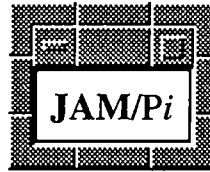
JAM version 4 applications must first be converted into version 5 applications before being transferred to **JAM/Pi**.

11.4

JAM VERSION 5 APPLICATIONS

Screens from **JAM** version 5.0 or later can be opened under **JAM/Pi**. 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.

<code>sm_drawingarea</code>	get the widget id (or handle) of the current JAM screen
<code>sm_translatecoords</code>	translate screen coordinates to display coordinates
<code>sm_widget</code>	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 "smpl.h"

void sm_adjust_area()
```

DESCRIPTION

This routine redisplay the current screen, recalculating 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.

<i>Scope</i>	<i>Description</i>
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

Pi/Windows

Pi/Motif

Pi/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.

<i>Scope</i>	<i>Description</i>
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

Pi/Windows

Pi/Motif

Pi/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 "mwin.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/Pi** 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

Pi/Windows

Pi/Motif

Pi/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 "smpl.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 text 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.

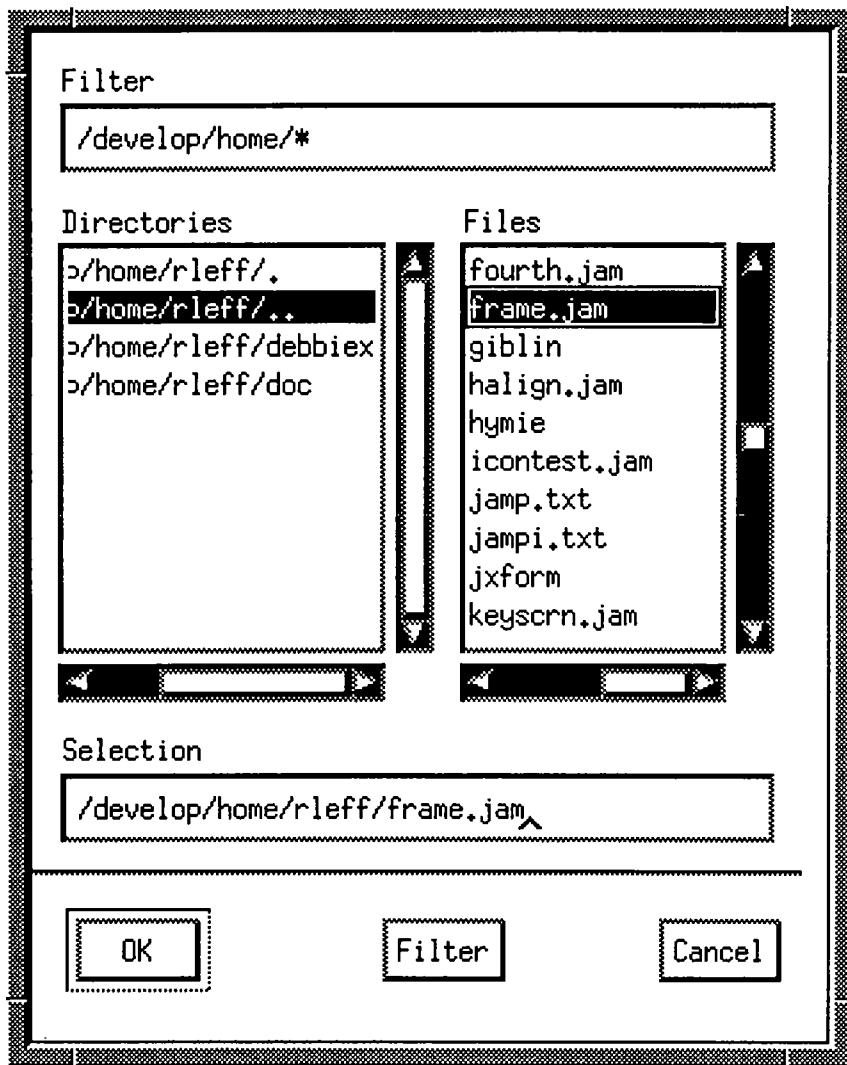


Figure 64: A Motif File Selection Box



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 <code>buffer</code> , and returns a one. It is up to the developer to provide a properly sized buffer. The buffer's size is indicated by the <code>length</code> 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 <code>buffer</code> , and the function returns zero. |

W

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 `Pi/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 `Pi/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".

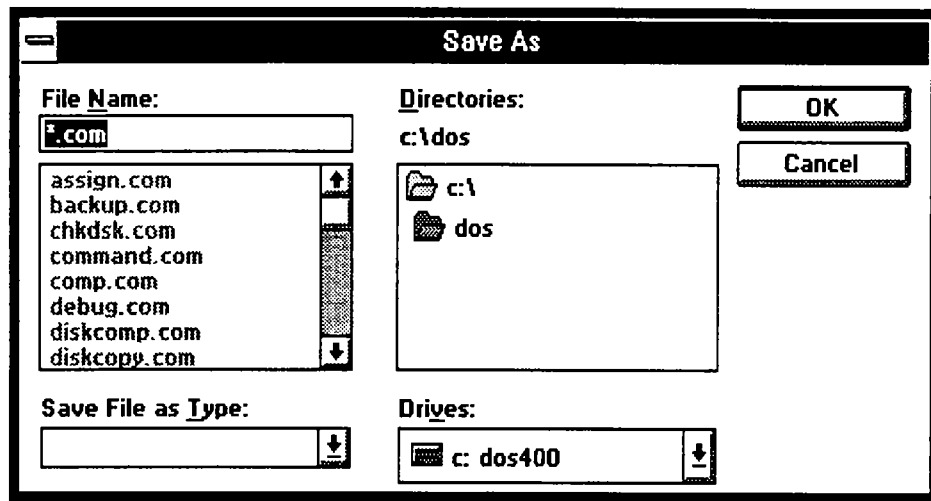


Figure 65: A Windows File Selection Box

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 "smpl.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 "smpl.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

Pi/Windows

EXAMPLE

```
#include "smdefs.h"
#include "smpl.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/Pi**, then the main routine should call `sm_skeyinit` before it calls `sm_menuinit`. The second library call will override the first in **JAM/Pi**, 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

Pi/Windows

Pi/Motif

Pi/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

Pi/Windows

Pi/Motif

Pi/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.

<i>Scope</i>	<i>Description</i>
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:

<i>Member</i>	<i>Description</i>
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.

<i>Member</i>	<i>Description</i>
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.

<i>Text Item Options</i>	<i>Value</i>	<i>Separator Options</i>	<i>Value</i>
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

Pi/Windows

Pi/Motif

Pi/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 "ssoftk.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 "ssoftk.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_mdelete( KS_APPLIC, "customer", count );  
  
...
```

sm_mnget

get menu bar item information

SYNOPSIS

```
#include "smsftk.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

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_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 be 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.

<i>Scope</i>	<i>Description</i>
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

Pi/Windows

Pi/Motif

Pi/OPEN LOOK

EXAMPLE

```
#include "smdefs.h"
#include "smsftk.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 "sm_pi.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/Pi** 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

Pi/Windows

Pi/Motif

Pi/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 Pi/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 Pi/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:

W

A handle to the widget.

M

O

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 "smpl.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

Pi/Motif
Pi/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/Pi 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 `SFTx` 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{*l*+1} or Row{*l*-1}, where *l* 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 `SFTn` 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.

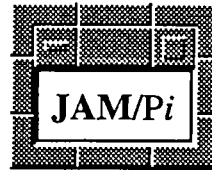
Corrective action: Check the spelling, presence, and permissions of the input file.

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 <code>jam.ini</code> and <code>win.ini</code> files are examples of initialization files. Contrast with <i>resource file</i> in Motif.
JAM/Pi	The JAM /Presentation <i>interfaces</i> 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 <i>interface</i> for Motif.
Pi/OPEN LOOK	The JAM /Presentation <i>interface</i> for OPEN LOOK.
Pi/Windows	The JAM /Presentation <i>interface</i> for Microsoft Windows.
resource file	A text file containing default specifications for the appearance and behavior of Motif applications. The <code>.Xdefaults</code> file, and the <code>XJam</code> file are examples of resource files. Individual items in the file are called resources. Contrast with <i>initialization file</i> 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 Pi/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 <i>widget</i> in favor of the term <i>control</i> .
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/Pi . Widgets are called <i>controls</i> in MS Windows. This document uses the term <i>widget</i> in favor of the term <i>control</i> .

INDEX

NOTE: Italicized page references (eg.— Array, *17*) indicate figures.

A

Alias, 158–160
 in bg extension, 83
 in fg extension, 83
 in font extension, 90
 sample
 Motif, 179
 OPEN LOOK, 189
 Windows, 162

Alignment, 23–37

Anchoring, 26–29

app—defaults directory, 148

Application mode, 23

Arranging screens, 23–37

Array, *18*, 18
 list box, 108–109
 scrolling
 behavior, 47–50
 optionmenu, 127
 spacing between elements, 33, 73, 140
 text editing in, 212

Attributes, 12–16
 application-wide, 13
 defaults, 145–190
 hierarchy, *13*
 JAM, 16
 lines and boxes, 58–61
 screen-wide, 15
 widget specific, 16–17

B

Background color
 resource in Motif/OPEN LOOK, 151
 screen, 55, 81–83
 widget, 65, 81–83

Base window, 163

bg, 55, 65, 81–83
 command line option in Motif/OPEN LOOK, 152

Bitmap, 68, 130
 compiling in Windows, 131
 height, 97
 icon, 104–106
 width, 97

Border, 42
 eliminating, 57, 116–117

Box, 84–86, 85
 color, 84
 creating, 57–61
 grid stretching and, 86
 layering, 86
 positioning, 36, 36–37
 style, 58, 84

box, 57–61, 84–86

Button. *See* Pushbutton; Togglebutton

C

Callbacks, 4

Character JAM
 converting applications, 216
 line drawing, 215
 portability to JAM/Pi, 4
 vs. JAM/Pi, 3

- Characters, 96
 - checkbox, 63, 87
 - Checklist, 20–21, 21, 63, 87
 - checkbox widget, 87
 - converting to list box, 108–109
 - togglebutton widget, 64, 143
 - Class
 - application, 147, 168, 181
 - widget, 168, 180
 - widgets for JAM fields, 171–172, 183–185
 - Colon expansion, 76
 - Color, 149–152
 - alias, 83, 158–160
 - background highlight on a PC, 215
 - box, 59, 84
 - frame, 67, 93
 - JAM colors, 149–150
 - line, 59, 99
 - ownColorMap resource in Motif and OPEN LOOK, 167, 180
 - palette, 81, 149–150
 - sample in Motif, 177
 - sample in OPEN LOOK, 189
 - sample in Windows, 162
 - push button, in Windows, 111
 - resources, 151
 - screen
 - background, 55, 81–83
 - foreground, 54, 81–83
 - widget
 - background, 65, 81–83
 - foreground, 65, 81–83
 - Combo box, 128
 - Command line, 14, 15, 148, 179–180
 - Motif, 167–168
 - bg switch, 152, 167
 - cascadeBug switch, 167
 - fg switch, 152, 167
 - fn switch, 154, 167
 - ind switch, 50
 - Command line (continued)
 - indicators switch, 168
 - name switch, 147
 - ownColormap switch, 167
 - setSensitive switch, 167
 - OPEN LOOK
 - bg switch, 152, 179
 - fg switch, 152, 179
 - fn switch, 154, 179
 - name switch, 147
 - ownColormap switch, 180
 - setSensitive switch, 180
 - Control. *See* Widget
 - Copy, 50–51, 212
 - Cursor
 - moving, 210–211
 - shape, 207–208
 - Cut, 50–51, 212
 - Cycle field, 64, 127
- ## D
- Data entry field, 17–18
 - multiline text widget, 113
 - text widget, 64, 141
 - Data entry mode, 44
 - Defaults, 7, 14, 15, 145–190
 - attributes, 12–16
 - dialog, 57, 88
 - Dialog box
 - for error messages, 44–47
 - icons, 46
 - screen extension, 57, 88
 - Display attributes. *See* Attributes
 - Display text, 17
 - placement, 28–29
 - Draw mode, 23, 25
 - Drawing area, 169, 182

E

Edit, 50–51

Elastic grid. *See* Grid

Error Message. *See* Message

Extensions, 53–74, 75–143

See also individual extensions by name

colon expansion of arguments, 76

field, 16, 61–74, 62, 78

array spacing, 33, 73, 140

background color, 65, 81–83

bitmap, 68, 130–133

checklist style togglebutton, 63, 87

disable grid adjustment, 33–34, 73, 115

font, 65, 89–91

foreground color, 65, 81–83

frame, 66, 92–93

horizontal anchor, 26–27, 73, 94–95

horizontal position, 34–35, 72, 102–103

in/out style togglebutton, 64, 143

label widget, 63, 107

list box, 63, 108–109

multiline button, 68, 111–112

multiline text widget, 63–64, 113–114

optionmenu, 64, 127–129

push button, 64, 136–137

radio style togglebutton, 64, 138

scale widget, 64, 139

suppress widget, 65–67, 126

text widget, 64, 141

vertical anchor, 27–28, 73, 94

vertical position, 34–35, 73, 102–103

widget height, 71, 96–97

widget type, 62–65

widget width, 72, 96–97

portability, 76

screen, 15, 54–61, 55, 80

background color, 55, 81–83

dialog box, 57, 88

draw a box, 57, 84–86

draw a line, 57, 98–101

eliminate title bar, 57, 125

Extensions, screen (continued)

font, 54, 89–91

foreground color, 54, 81–83

mouse pointer, 56, 134–135

pointer shape, 56, 134–135

prevent iconification, 43, 57, 122

prevent maximization, 57, 119

prevent resizing, 57, 124

specify icon, 43, 54, 104–105

start as icon (minimized), 43, 57, 106

start maximized, 57, 110

suppress border, 57, 116–117

suppress close, 57, 118

suppress move option, 57, 123

suppress window menu, 57, 120–121

title, 54, 142

summary tables, 78–81

syntax, 76

vs. resources, 75

F

fg, 54, 65, 81–83

command line option in Motif/OPEN
LOOK, 152

Field

See also Array; Group

cycle, 127

data entry, 17–18

multiline text widget, 113

justification and positioning, 24, 94

menu, 19–20, 136–137

non-display, 126

protected, 17, 107

label widget, 107

non-display, 126

scrolling behavior, 47–50

shifting behavior, 47–50

Field extensions. *See* Extensions

File selection box, 226–230

file types list, 231–232

Focus, 41–42

mouse, 209–210

Font, 153–157
 alias, 90, 158–160
 application default, 153–154
 field extension, 89–91
 fixed width, 30–32, 31
 fn command line option in Motif/OPEN LOOK, 154
 font resource in OPEN LOOK, 179
 fontList resource in Motif, 167
 location, 153–154
 naming, 155–160
 proportional, 30–32, 31
 and shifting fields, 48
 screen, 154
 screen extension, 89–91
 widget's, 154
 xfontsel, 157

font, 54, 65, 89–91

font resource in OPEN LOOK, 179

fontList resource in Motif, 167

Foreground color
 resource in Motif/OPEN LOOK, 151
 screen, 54
 widget, 65

formMenus, 164

Frame, 66–68, 92–93, 93
 color, 67, 93
 MDI, 40
 style, 66, 92
 vs. box, 92

frame, 66–68, 92–93

G

Greyed text, 167, 180

Grid, 23–25, 24
 boxes and, 86
 disabling stretching, 33, 73, 115
 equally spacing array elements, 73, 140

Grid (continued)
 font and grid size, 30–32
 lines and, 100
 separators, 37
 units, 97

Group, 20–21
 creating a checkbox widget, 87
 creating a list box, 63, 108
 creating a radiobutton widget, 138
 creating a togglebutton widget, 143

GUI independent fonts and colors. *See* Alias

GUI interface routines, 217, 224–225
 sm_drawingarea, 224–225
 sm_translatecoords, 252–254
 sm_widget, 255–256

GUI library, 213–214

H

halign, 26–27, 27, 73, 94–95
 and whitespace, 29–30

height, 71, 96–97

hline, 57–61, 98–101

hoff, 34–35, 72, 102–103

Horizontal alignment. *See* halign

Horizontal positioning. *See* hoff

I

icon, 43, 54, 104–105

Iconification, 43, 54, 57, 104–105, 106
 preventing, 122

iconify, 57, 106

Inches, 97

Indicators, 49–50, 168
 name, Motif, 171

Initialization file, 7, 14, 160–162
 aliases, 158–160
 color aliases, 158–160
 colors, 149–152
 font, 153–157
 FrameTitle, 160
 GrayOutBackgroundForms, 160
 JAM Colors, 149
 JAM ColorTable, 158
 JAM Fonts, 153
 JAM FontTable, 158
 JAM Options, 160–161
 location, 148
 name, 145
 sample, 162
 SMTERM, 161
 StartupSize, 160
 StatusLineColor, 161
 syntax, 146–147

Item selection screen, 127, 129

J

jam.ini, 14, 146
 sample, 162

 jmain.c, 145

 JPL comments. *See* Extensions

 Justification, 24, 26, 94

 jxmain.c, 145

K

Keysets, 51–52
 kset2mnu, 261–262

 Keytops, 47

 kset2mnu utility, 261–262

L

label, 63, 107

 Label widget, 17, 17, 107, 107
 bitmap, 130–133
 creating, 63
 name
 Motif, 171
 OPEN LOOK, 184

 LDB, optionmenus and, 128

 Left justified, 24, 26, 94

 Library routines, 217–257
 file selection box, 218
 GUI interface routines, 217
 menu bar, 217–218
 sm_adjust_area, 35, 219
 sm_c_menu, 220–221
 sm_d_menu, 222–223
 sm_drawingarea, 170, 183, 213, 224–225
 sm_filebox, 226–230
 sm_filetypes, 231–232
 sm_menuinit, 233
 sm_mn_forms, 234
 sm_mnadd, 235–237
 sm_mnchange, 238–239
 sm_mndelete, 240–241
 sm_mnget, 242–243
 sm_mninsert, 244–245
 sm_mnitems, 246–247
 sm_mnnew, 248–249
 sm_r_menu, 250–251
 sm_translatecoords, 213, 252–254
 sm_widget, 171, 183, 213, 255–256
 sm_win_shrink, 257–258
 sm_X11init, 145

 Line, 98, 98–101
 color, 99
 creating, 57–61
 layering, 100
 positioning, 36, 36–37
 style, 58, 99

 Line drawing characters, 215

 list, 63, 108–109

List box, 20–21, 21, 108–109, 109
 creating, 63
 height, 97
 name
 Motif, 172
 OPEN LOOK, 184
 vertical anchor, 109

Look and feel, 2

M

maximize, 57, 110

MDI, 40–41, 41
 dialog boxes, 88
 icon location in, 43
 maximized window, 110

Menu, 19–20, 136–137
 See also Menu bar
 selecting, 210–211

Menu bar, 191–205, 200
 add an item, 235–237
 alter an item, 238–239
 cascadeBug resource in Motif, 192
 cascadeBug resource in Motif, 167
 close, 220–221
 converting keysets into, 205, 261–262
 create new menu bar, 248–249
 delete an item, 240
 display, 222–223
 edit heading, 50–51
 enabling support, 202
 formMenus resource, 164, 193
 get data about, 242–243
 get number of items, 246–247
 initialize support, 233
 insert an item, 244–245
 install in memory, 234
 installing, 202–203
 library routines, 201–202, 217–218
 See also Library routines
 prototyping, 202
 location, 164, 191–192

Menu bar (continued)
 menu2bin utility, 259–260
 mouse, 209
 pop-up, 192, 209
 read and display, 250–251
 scope, 164, 192–193
 script, 194–200
 comments, 198
 converting to binary, 259–260
 display options, 196–197
 general structure, 194
 global display options, 198
 keywords, 194–198
 sample, 198–200
 storing in memory, 203
 testing, 200–201
 vs. softkeys, 52, 204–205
 widget hierarchy in Motif, 173–175
 widget hierarchy in OPEN LOOK,
 185–187
 window heading, 41, 210

Menu mode, 44

menu2bin utility, 259–260

Message
 error, 44–47
 dialog box icons, 46
 optionmenu limitation, 129
 status, 44
 formStatus resource, 163

Millimeters, 97

Mode, menu vs. data entry, 44

Motif
 color naming, 150
 font naming, 155–157
 overstrike mode, 213
 resources. *See* Resource file
 shift/scroll indicators, 50

Mouse, 207–212
 buttons, 208
 editing text, 212
 focus, 209–210
 in select mode, 212
 menu bars, 209
 move function, 210

Mouse (continued)
 offset function, 210
 pointer shape, 56, 134–135, 207
 resize function, 210
 scrolling, 49, 211–212
 selecting text, 50
 shifting, 49, 211–212
 toggling mode with, 44

MS Windows. *See* Windows

multiline, 68, 111–112

Multiline text widget, 113–114, 114
 creating, 63
 name
 Motif, 171
 OPEN LOOK, 184

Multiple Document Interface. *See* MDI

multitext, 63, 113–114

N

noadj, 33–34, 35, 73, 115
 noborder, 57, 116–117
 noclose, 57, 118
 nomaximize, 57, 119
 nomenu, 57, 120–121
 nominimize, 43, 57, 122
 nomove, 57, 123
 Non-display field, 126
 noresize, 57, 124
 notitle, 57, 125
 nowidget, 65, 126

O

OLJam file, 146
 sample, 188–190

OPEN LOOK
 color naming, 150
 font naming, 155–157
 overstrike mode, 213
 resources. *See* Resource file
 shift/scroll indicators, 50

optionmenu, 64, 127–129

Optionmenu widget, 127–129, 129
 creating, 64
 height, 97
 name
 Motif, 172
 OPEN LOOK, 184
 populating, 70–71

P

Paste, 50–51, 212

Pixels, 96

pixmap, 68, 130–133

pointer, 56, 134–135

Pop-up menu bar, 192

Portability, 4

Positioning, 23–37
 boxes, 86
 lines, 100–101

Protected field, 17, 107
 non-display, 126

Push button, 19, 19–20, 136–137, 137
 bitmap, 130–133
 color in Windows, 19, 111
 creating, 64
 multiline, 68, 111–112, 112
 name
 Motif, 171
 OPEN LOOK, 184
 selecting, 210–211
 text alignment in Motif, 20, 21
 toggling into menu mode, 44

pushbutton, 64, 136–137

R

Radio button, 20–21, 21, 64, 138
 converting to list box, 108–109
 radiobutton widget, 138
 togglebutton widget, 64, 143

radiobutton, 64, 138

Range check, 139

Resource. *See* Resource file

Resource file, 7, 14, 15, 163–173
 aliases, 158–160
 armPixmap, 131
 background, 167, 179
 vs. bg extension, 82
 background resource, 152
 baseWindow, 47, 52, 163, 168, 181
 bitmaps in Windows, 131
 cascadeBug in Motif, 167, 192
 class name, 147
 color aliases, 158–160
 colors, 149–152
 focusAutoRaise, 42, 165
 font, 153–157
 font resource in OPEN LOOK, 179
 fontList, 167
 foreground, 167, 179
 foreground resource, 152
 formMenus, 52, 193
 formStatus, 47, 163
 indicators, 50, 168
 location, 148
 Motif, 167–179
 sample, 175–179
 names, 145–146
 OPEN LOOK, 179–190
 sample, 188–190
 overriding extensions, 75, 151
 ownColorMap, 167, 180
 restricting resources to a screen, 170, 182
 screen title, 39, 142
 selectPixmap, 131
 setSensitive, 167, 180
 syntax, 146–147

RGB, 149

rgb.txt, 166

Right justified, 24, 26, 94

S

scale, 64, 139

Scale widget, 139, 139
 accessing data in, 139
 creating, 64
 name
 Motif, 172
 OPEN LOOK, 184
 range, 69

Scope, 192–193

Screen
 appearance, 39–44
 arrangement, 23–37
 fine tuning, 33–35
 border, 42
 eliminating, 57, 116–117
 decorations, 56–57
 focus, 41–42
 mouse, 209–210
 font, 89–92
 handle, 224–225
 iconification, 43
 minimizing, 43
 mouse pointer shape, 134–135
 moving, 210
 refresh, 35, 219
 resizing, 210
 resources
 Motif, 170
 OPEN LOOK, 182
 scroll bar, 39
 scrolling, 210
 size, 257–258
 size and fonts, 30–32
 start maximized, 110
 title bar, 39, 142
 suppressing, 57
 trim, 257–258

Screen (continued)
 widget hierarchy
 Motif, 169
 OPEN LOOK, 182
 widget id, 224–225

Screen extensions. *See* Extensions

Script. *See* Menu bar

Scroll bar
 list box, 69, 108
 multiline text widget, 69, 113
 scrolling with mouse, 211–212

Scrolling array, 47–50
 list box, 108
 multiline text widget, 113

Scrolling indicator, 49–50

Select mode, 212

Separator, 98–101
 creating, 57–61
 positioning, 36, 36–37, 100–101

SFTS, 200–201

Shifting field, 47–50
 shifting with mouse, 211–212

Shifting indicator, 49–50

Sibling window, 43
 mouse, 209

sm_.... *See* Library routines

SMTERM, 161

Soft keys, 51–52

space, 33, 34, 73, 140

SPF11, 53

SPF12, 53

State abbreviations, 97

Status line, 44–47
 formStatus resource, 163
 location, 47, 163

System command, 214

T

Test mode, 23, 25

Text
 cut, copy and paste, 50–51
 editing with mouse, 212

text, 64, 141

Text widget, 17–18, 18, 141, 141
 creating, 64
 editing text, 212
 multiline, 63, 113–114
 height, 97
 name
 Motif, 171
 OPEN LOOK, 184
 shifting, 48
 toggling into data entry mode, 44

title, 54, 142

Title bar, 39
 suppressing, 57, 125
 text, 54, 142

Togglebutton, 20–21, 21
 bitmap, 130–133
 multiline, 111–112, 112
 name
 Motif, 171
 OPEN LOOK, 184
 selecting, 210–211

togglebutton, 64, 143

U

Units of measurement, 60, 96–97

Utilities, 258–262
 kset2mnu, 261–262
 menu2bin, 259–260

V

valign, 27–28, 73, 94–95

Vertical alignment. *See* valign

Vertical positioning. *See* voff

vline, 57–61, 98–101

voff, 34–35, 73, 102–103

VWPT key, 210

W

Whitespace, 23, 29–30

Widget, 11–21

See also individual widgets by name

adjusting position, 34–35

anchoring. *See* Anchoring

attribute hierarchy, 13

attributes, 12–16

default type, 11

drawing area, 170, 183

expanding into whitespace, 29–30

font, 65, 89–92

forcing a type, 61, 65

handle, 255–256

hierarchy

Motif, 168–175

base screen, 168–169

boxes, 173

dialog box, 169

display text, 173

fields, 171–172

JAM screens, 169–171

lines, 173

menu bars, 173–175

OPEN LOOK, 180–187

base screen, 181

boxes, 185

dialog box, 183

display text, 185

fields, 183–185

JAM Screens, 182–183

lines, 185

menu bars, 185–187

id, 255–256

invisible, 65

JAM objects into, 17–21

names in Motif, 168–175

names in OPEN LOOK, 180–187

placement, 26–29

horizontal, 72, 102–103

vertical, 73, 102–103

recalculating position, 35

scroll bars, 69, 108, 113

setting the type, 62

size

default, 32

specifying height, 71, 96–97

specifying width, 72, 96–97

width, 72, 96–97

win.ini, 14, 161

Windows

color naming, 149

control panel, 14, 149, 161

font naming, 155

maximized frame, 40

MDI, 40–41

Multiple Document Interface, 40–41

system commands, 214

title bar, 40

X

XAPPLRESDIR, 148

Xdefaults, 14, 15, 148

sample, 175–179, 188–190

xfonstsel, 157

XJam file, 146

sample, 175–179

xlsfonts, 155

xoff. *See* hoff

xrdb, 148

Y

yoff. *See* voff