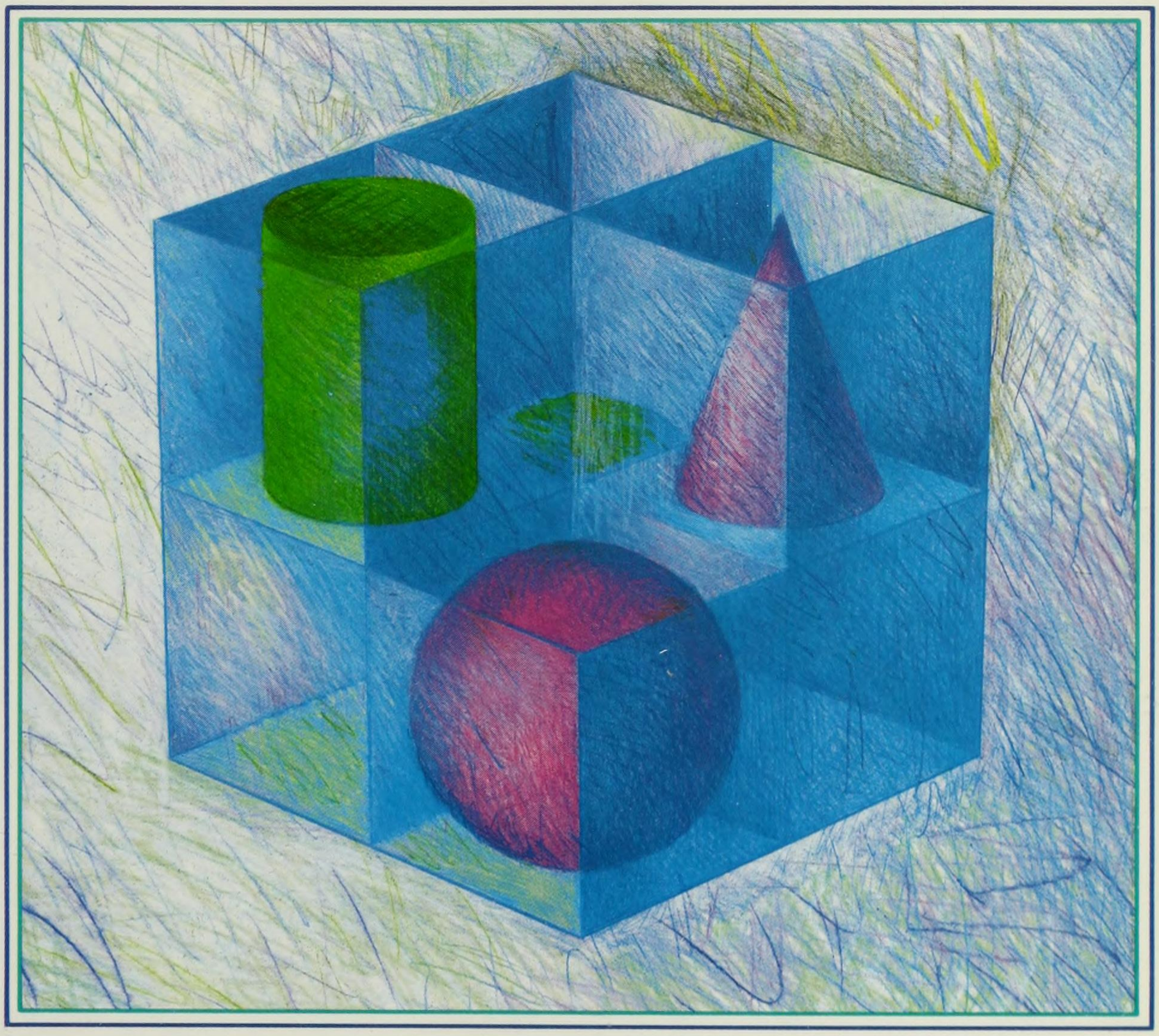
OSF/Motif™

Release 1.1
Revised and Updated

Style Guide



OPEN SOFTWARE FOUNDATION

Digitized by the Internet Archive
in 2012

http://archive.org/details/osfmotifstylegui00open

OSF/Motif™

Style Guide

Revision 1.1

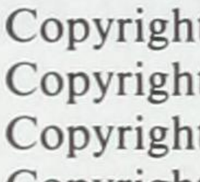
(For OSF/Motif Release 1.1)

Open Software Foundation

Prentice Hall, Englewood Cliffs, New Jersey 07632

Cover design
and cover illustration: **BETH FAGAN**

This book was formatted with troff



Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

The information contained within this document is subject to change without notice.

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

All rights are reserved. No part of this publication may be photocopied, reproduced, or translated into another language without the prior written consent of the Open Software Foundation, Inc.

Copyright © 1991, 1990, Open Software Foundation, Inc.
Copyright © 1989 Digital Equipment Corporation
Copyright © 1987, 1988, 1989 Hewlett-Packard Company
Copyright © 1988 Massachusetts Institute of Technology
Copyright © 1988 Microsoft Corporation
ALL RIGHTS RESERVED

FOR U.S. GOVERNMENT CUSTOMERS REGARDING THIS DOCUMENTATION AND THE ASSOCIATED SOFTWARE

Notice: Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in the FAR Section 52.227-19 "Computer Software - Restricted Rights."

Unpublished - All rights reserved under the Copyright Laws of the United States.

This notice shall be marked on any reproduction of this data, in whole or in part.

Open Software Foundation, OSF, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.
DEC and DIGITAL are registered trademarks of Digital Equipment Corporation.
Microsoft is a registered trademark of Microsoft Corporation.
Presentation Manager is a trademark of International Business Machines Corporation.
X Window System is a trademark of the Massachusetts Institute of Technology.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-640616-5

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia PTY. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Preface ix
Audience x
Contents x
Typographic Conventions xi
Compliance Conventions xiii
Chapter 1 User Interface Design Principles 1-1
1.1 Adopt the User's Perspective 1-2
1.2 Give the User Control 1-3
1.3 Use Real-World Metaphors 1-4
1.4 Keep Interfaces Natural 1-6
1.5 Keep Interfaces Consistent 1-8
1.6 Communicate Application Actions to the User 1-10
1.7 Avoid Common Design Pitfalls 1-12
Chapter 2 Input and Navigation Models 2-1
2.1 The Keyboard Focus Model 2-2
2.2 The Input Device Model 2-5
2.3 The Navigation Model 2-13
Chapter 3 Selection and Component Activation 3-1
3.1 Selection Models 3-2
3.2 Selection Actions 3-15
3.3 Component Activation 3-23
Chapter 4 Application Design Principles 4-1
4.1 Choosing Components 4-2

4.2 Layout 4-12
4.3 Interaction 4-45
4.4 Component Design 4-52
Chapter 5 Window Manager Design Principles 5-1
5.1 Configurability 5-2
5.2 Window Support 5-2
5.3 Window Decorations 5-6
5.4 Window Navigation 5-14
5.5 Icons 5-15
Chapter 6 Designing for International Markets 6-1
6.1 Collating Sequences 6-2
6.2 Country-Specific Data Formats 6-3
6.3 Icons, Symbols, and Pointer Shapes 6-7
6.4 Scanning Direction 6-8
6.5 Designing Modularized Software 6-9
6.6 Translating Screen Text 6-9
Chapter 7 Controls, Groups, and Models Reference Pages 7-1
Accelerators 7-4
Activation 7-5
Basic Controls 7-8
Basic Groups 7-10
Canvas 7-12
CascadeButton 7-13
CheckButton 7-15
CommandBox 7-18
CommandDialog 7-20
Composition 7-21
Default Activation 7-22
DialogBox 7-23
Drag and Drop 7-26
Edit Menu 7-28
ErrorDialog 7-31

Field Controls 7-32
File Menu 7-35
FileSelectionBox 7-38
FileSelectionDialog 7-41
Focus 7-43
Frame 7-45
Framing Groups 7-46
Help Menu 7-47
Icon Menu 7-49
IconBox 7-50
Icons 7-52
InformationDialog 7-54
Input Devices 7-55
Label 7-57
Layout Groups 7-58
List 7-59
MainWindow 7-63
MenuBar 7-65
Menus 7-69
MessageDialogs 7-71
Mnemonics 7-72
Navigation 7-73
OptionButton 7-76
PanedWindow 7-79
Panel 7-81
Pointer Shapes 7-83
Primary Selection 7-86
PromptDialog 7-88
PushButton 7-90
QuestionDialog 7-92
Quick Transfer 7-94
RadioButton 7-96

Sash 7-99
Scale 7-103
ScrollBar 7-107
ScrolledWindow 7-111
Selection 7-113
SelectionBox 7-115
SelectionDialog 7-117
Separator 7-119
Text 7-120
ToggleButton 7-127
WarningDialog 7-129
Window Menu 7-130
WorkingDialog 7-133

Appendix A Common Bindings for Virtual Buttons and Keys . . . A-1

Appendix B OSF/Motif Widgets and Components Correspondence B-1

Glossary GL-1

List of Figures

Figure 2-1. A Box Style Location Cursor 2-16
Figure 2-2. An Outline Highlight Style Location Cursor 2-17
Figure 2-3. Text Cursor Shapes 2-17
Figure 2-4. A Text Cursor Style Location Cursor 2-18
Figure 2-5. An Item Cursor Style Location Cursor 2-18
Figure 2-6. A Fill Style Location Cursor 2-19
Figure 4-1. A Typical MainWindow and Its Common Client Areas 4-14
Figure 4-2. The File Menu and Its Selections 4-18
Figure 4-3. The Edit Menu and Its Selections 4-21
Figure 4-4. The Help Menu and Its Selections 4-23
Figure 4-5. A CommandDialog 4-25
Figure 4-6. A FileSelectionDialog 4-26
Figure 4-7. A PromptDialog 4-27
Figure 4-8. A SelectionDialog 4-28
Figure 4-9. An ErrorDialog 4-30
Figure 4-10. An InformationDialog 4-30
Figure 4-11. A QuestionDialog 4-31
Figure 4-12. A WarningDialog 4-32
Figure 4-13. A WorkingDialog 4-32
Figure 4-14. A Default PushButton 4-47
Figure 4-15. A Disabled Menu Element 4-49
Figure 5-1. A Typical OSF/Motif User Environment 5-3
Figure 5-2. A Typical OSF/Motif Window Layout 5-7

Figure 5-3. The Window Menu Button with Menu Pulled Down 5-12
Figure 5-4. A Typical OSF/Motif Icon 5-16
Figure 5-5. A Typical OSF/Motif Icon Box 5-17

List of Tables

Table 2-1. Keyboard Navigation Bindings 2-5
Table 2-2. Window Navigation Bindings 2-20
Table 2-3. Field Navigation Bindings 2-22
Table 2-4. Component Navigation Bindings 2-24
Table 2-5. Component Navigation Bindings 2-25
Table 2-6. Menu Traversal Bindings 2-27
Table 2-7. Menu Navigation Bindings 2-29
Table 2-8. Scrolled Region Navigation Bindings 2-31
Table 3-1. Range Selection Bindings 3-7
Table 3-2. Keyboard Selection Bindings 3-11
Table 3-3. KSelectAll and KDeselectAll Bindings 3-14
Table 3-4. KReselect Bindings 3-18
Table 3-5. Primary Transfer Bindings 3-19
Table 3-6. Quick Transfer Bindings 3-21
Table 3-7. Clipboard Action Bindings 3-23
Table 3-8. Activation Bindings 3-25
Table 3-9. KHelp Bindings 3-27
Table 4-1. Suggested Window Types 4-9
Table 5-1. KWindowMenu Bindings 5-12
Table 5-2. Window Navigation Bindings 5-15
Table 5-3. Icon Box Traversal Bindings 5-20
Table A-1. Virtual Button Bindings for a 3-Button Mouse A-1
Table A-2. Virtual Button Bindings for a 2-Button Mouse A-1
Table A-3. Virtual Button Bindings for a 1-Button Mouse A-2

Table A-4. Virtual Modifier Bindings A-3
Table A-5. Virtual Key Bindings A-3
Table B-1. Component to Widget Correspondence B-1

Preface

The *OSF/Motif Style Guide* provides a framework of behavior specifications to guide application developers, widget developers, user interface system developers, and window manager developers in the design and implementation of new products consistent with the OSF/Motif™ user interface. This *Style Guide* is also closely consistent with Microsoft Windows, Presentation Manager, and Common User Access (CUA).

The *Style Guide* establishes a consistent behavior among new products by drawing out the common elements from a variety of current behavioral models. The *Style Guide* anticipates the evolution of graphical user interfaces as new technology becomes available and as the use of the Motif™ user interface spreads. Behavioral guidelines will be added over time as they become stable.

For specific details of coding the implementation into an application program, widget, or window manager see the other volumes of the OSF/Motif documentation set.

Audience

This document is written for four audiences. The following text suggests the sections in this guide that are relevant to each audience. We recommend that you read through the entire *Style Guide* once to familiarize yourself with all user interface design concepts and to ensure that you do not miss anything.

- Application Designers

  Should be familiar with the contents of Chapters 1, 4, and 6.

- Widget Designers

  Should be familiar with the contents of Chapters 1, 2, 3, 4, 6, and 7.

- User Interface System Designers

  Should be familiar with the entire contents of this guide.

- Window Manager Designers

  Should be familiar with the contents of Chapters 1, 4, 5, and 6.

Contents

This document is organized into seven chapters and two appendixes.

- *Chapter 1* provides general user interface design principles. Everyone should read this chapter.
- *Chapter 2* describes the input and navigation models. New widget designers and user interface system designers should read this chapter.
- *Chapter 3* describes the selection and activation models. New widget designers and user interface system designers should read this chapter.
- *Chapter 4* describes user interface component choices, layout, and interaction. Everyone should read this chapter.

- *Chapter 5* describes window manager design. Window manager designers and user interface system designers should read this chapter.
- *Chapter 6* introduces and briefly describes internationalization and localization concepts and issues as they relate to user interface design. Everyone should read this chapter.
- *Chapter 7* provides reference information for concepts described in earlier chapters, and provides detailed information about components. This chapter should be read by new widget designers and can by used as a reference by everyone.
- *Appendix A* provides default keyboard and mouse bindings.
- *Appendix B* shows the correspondence between OSF/Motif widgets and components described in this *Style Guide*.

Typographic Conventions

The Motif document set uses the following typographic conventions:

- **Boldfaced** strings represent key and button bindings, and literal values. **Boldfaced** strings also represent the first use of words described in the glossary.
- **<Key>** represents a key on the keyboard. The word in italics corresponds to the engraving on the actual key.
- Components of the user interface are represented by capital letters on each major word in the name of the component, such as PushButton.

See "Compliance Conventions" below for an explanation of the asterisks (*) that appear in the margins.

Since not all keyboards are the same, it is difficult to give style guidelines that are correct for every manufacturer's keyboard. To solve this problem, this guide describes keys and buttons using a **virtual binding** mechanism, by the action they perform. Everywhere that keyboard input is specified, the keys are indicated by their virtual key name. This guide separately explains which

actual keys are commonly used for each virtual key. For instance, the virtual key **KActivate** is used for keyboard activation of components and is commonly bound to **<Return>** and **MCtrl <Return>**, or **<Enter>** and/or **MCtrl <Enter>** depending on the keyboard. This mechanism also simplifies the descriptions of behavior where multiple keys can be bound simultaneously to one virtual key.

The virtual key mechanism includes keys, modifiers, and mouse buttons. Virtual mouse buttons start with the letter **B**. For a 3-button mouse, the leftmost mouse button is usually defined as **BSelect**, the middle mouse button is usually defined as **BDrag**, and the rightmost mouse button is usually defined as **BCustom**. Details about how virtual mouse buttons are usually defined are described in Section 2.2, and where a new virtual button is described for the first time.

All normal virtual keys start with the letter **K**. All modifier virtual keys start with the letter **M**.

This guide assumes that a keyboard has all the standard alphabetic, numeric, and symbol keys, as well as the special keys corresponding to the actions: Enter, usually **<Return>** or **<Enter>** Tab, Spacebar, Backspace, Delete, Escape, Insert, ↑, ↓, ←, →, Begin Line, usually **<Begin>** or **<Home>**, End Line, usually **<End>**, Page Up, and Page Down. Each of these keys must be available either as specified or * using other keys or key combinations if the specified key is * unavailable.

This guide also requires the nominal keys corresponding to the actions: Move to the MenuBar (**<F10>**), pop up a menu (**<Menu>** or **<F4>**), provide help (**<Help>** or **<F1>**), toggle add mode (**<Shift> <F8>**), and move to the next pane (**<F6>**). Each of this * keys must be available either as specified or using other keys or key * combinations if the specified key is unavailable. If the chosen * binding involves a function key, the one listed above must be used, * unless it is not available on the keyboard. Additional keys, including navigation keys and special keys like **<Menu>** and **<Help>**, are suggested, but not necessary for full keyboard equivalence.

This guide also assumes that all keyboards have the three modifier keys **MShift**, **MCtrl**, and **MAlt**, which are usually bound to **<Shift>**, **<Ctrl>**, and **<Alt>**. Your keyboard may have the virtual

buttons, keys, or modifiers bound as appropriate. Appendix A lists all of the virtual bindings used in this guide, as well as suggested and alternative bindings. To avoid confusion, this guide also mentions the suggested bindings of virtual keys throughout.

Compliance Conventions

Throughout the *Style Guide* "must," "should," and "can" have special meanings. Guidelines with "must" in them are requirements for *Style Guide* compliance. Any guideline with "must" will be included in the *OSF/Motif Level 1 Certification Checklist* for *Style Guide* compliance. Any guideline with "must" will be marked in the margin with an asterisk. Guidelines with "should" in them are recommendations. We consider them important for interapplication consistency, but we do not want to require them for compliance. You should follow them as closely as you are able. Guidelines with "can" in them indicate optional elements of user interface style.

The process for how *Style Guide* elements migrate from options to requirements is described in the *OSF/Motif Level 1 Certification Checklist* .

Note that by default this guide assumes your application is being designed for a left-to-right language direction environment, and that the application is written in English. Many of these guidelines can, and in fact should, be modified based on both language and scanning direction.

Chapter 1

User Interface Design Principles

A user interface is simply the interface between an application and the user of an application. The primary goal of a user interface is to help user interface designers easily create applications that increase user effectiveness and satisfaction. By following the guidelines presented in this *Style Guide*, you can create applications that are well designed and easy to use. These guidelines pertain to all sorts of applications, from spreadsheets and word processors to CAD tools.

To be effective, a user interface allows a user to interact with an application simply and naturally. Successful user interface designers keep the user in mind while designing an application. Keep the user in mind by following these two principles:

- Know the user.
- Empower the user.

The user of an application, above all, wants to get the job done. A user interface needs to be designed so the user can quickly and easily complete the tasks. Users want mastery over an application. Make it simple for the user to master the basics of your application. At the same time, you can include advanced methods of interaction with the application, shortcuts for the user. Users are curious and exploratory. They will find these shortcuts as they use and master

the application. Such shortcuts need not be as intuitive as the regular methods of interaction.

This chapter discusses the following guidelines for creating user interfaces that are consistent and easy to use. Because of the particular nature of your client application, component, or window manager, or in response to customers' needs, you may not be able to apply all of these principles all of the time.

- Adopt the user's perspective.
- Give the user control.
- Use real-world metaphors.
- Keep interfaces natural.
- Keep interfaces consistent.
- Communicate application actions to the user.
- Avoid common design pitfalls.

1.1 Adopt the User's Perspective

Effective design starts with adopting the user's point of view, which is often difficult to do. Application designers tend to see an application as the implementation of functions. In contrast, the user sees an application in terms of its interface.

Good design is rooted in an understanding of the user's work. A well-designed application solves users' problems, makes their work easier, and offers them new capabilities. The two most effective ways to understand the user's work are to involve users in the design, and to be a user yourself.

Input from users can help determine both appropriate functions and methods for presenting them. Involve users as early as possible in the design process because as the design progresses and the schedule closes in, the possibilities for design change decrease.

You do not need a working prototype to involve users. In fact, you can even involve users while you are writing specifications. At this stage, you can watch users work in order to understand the

environment in which your application will be used. Talk to these users about their work, their current tools, and their goals for new tools.

For example, if you are designing software to create and display charts and graphs during meetings, you might attend meetings at various customer sites, see how charts and graphs are currently used in meetings, and interview meeting participants to learn what they would like to see in a new tool. Once you have a working prototype of your application, invite users to test it to see if your interface meets the goals you established for it.

Try to use your application in real situations. Using an application can provide critical insights into user interface problems. Acquiring experience with the application can be difficult and time consuming, but is a worthwhile exercise. Before you even create the interface for your application, you can use similar applications, even competitive products, to help you understand the user's tasks.

1.2 Give the User Control

Users want and need to be in control of the tools they use to perform their work. The user can be in control when an application is flexible, and uses progressive disclosure.

1.2.1 Keep Interfaces Flexible

Providing multiple ways for users to access application functions and accomplish their tasks increases their sense of control. Flexibility enables users to select the best method of accessing a function based on the criteria they choose: experience level, personal preference, unique situation, or simply habit. For example, a user can access a function through a PullDown Menu, direct manipulation of an object, a mnemonic key press, or a keyboard accelerator.

Your application should also be configurable. Allowing users to configure settings and select personal preferences enhances their sense of control and encourages them to take an active role in

understanding your product and how it works. To be effective, the configurability of your application needs to be easily accessible.

1.2.2 Use Progressive Disclosure

Design your application so that the necessary and common functions are presented first and in a logical order. Make the more sophisticated and less frequently used functions hidden from immediate view, but still available. For example, use a DialogBox to hide settings that are not accessed often.

Decisions about the placement of functions are not easy to make. From the implementation standpoint, all functions are important. Often, however, a relatively small number of functions account for the majority of use. Make sure that these important functions are prominently featured in the presentation of the interface. Also remember that they can be prominent only if other functions are hidden.

1.3 Use Real-World Metaphors

A good user interface allows the user to transfer skills from real-world experiences. For example, PushButtons push, and Scales slide. This makes it easier for the user to infer how to use an application. When you design a new component, consider how a similar real-world control performs to incorporate the metaphor into the new component. Real-world metaphors can extend to groups of components as well, especially when making a computer-based user interface to replace a mechanical user interface.

1.3.1 Allow Direct Manipulation

Users need to be able to directly manipulate elements of the user interface and their applications. For example, the user needs to be able to directly scroll Text with a ScrollBar, rather than using a keyboard-driven command. Direct manipulation simulates the real world where the user employs tools to perform tasks on physical objects. Users control applications by directly manipulating graphical components similar to real-world controls, rather than entering a command on a command line. Direct manipulation reduces the amount of information the user needs to memorize.

Direct manipulation connects an action to an observable response from a component. Using direct manipulation, the user gets an immediate visible result from each action.

The direct manipulation model is an object-action model. That is, you first select an object or group of objects, then you perform an action on the selected objects. An object-action model allows the user to see what elements will be acted on before performing an action. It also allows multiple actions to be performed successively on the selected elements.

Although it is important to allow direct manipulation of the objects in your application, you must also support methods for interacting * with your application by keyboard-only users. These methods can also be used by advanced users to perform some tasks more quickly.

1.3.2 Provide Rapid Response

Make your application respond to input as rapidly as possible. The immediacy of the visual response is crucial to the experience of direct manipulation. When using components, provide the application's response immediately and in proportion to the component's actions. The application must also have a consistent * speed of response. Delays, disproportionate responses, or inconsistent responses can render an otherwise well-designed application unusable. Performance problems make it difficult for the user to concentrate on the task at hand.

1.3.3 Provide Output as Input

Another feature of direct manipulation is that the output of one part of an application or the output of the application itself is also available as input. For example, if one action produces a list of file names, another action can select them for use elsewhere.

The user manipulates objects by locating them and clicking on them rather than typing in their names. Design so that the only time the user needs to type a name is to create an object. A well-designed application reduces the amount of information the user needs to memorize to perform tasks.

1.4 Keep Interfaces Natural

You can extend the concepts of giving the user control and using real-world metaphors to arrange your application so that tasks flow naturally. Users need to be able to anticipate the natural progression of each task; through this anticipation, they are able to complete tasks more quickly.

Each screen object needs to have a distinct appearance that the user can easily recognize and quickly understand. At the same time, the style of the interface needs to graphically unify these elements and ensure a consistent and attractive appearance at any screen resolution.

1.4.1 Make Navigation Easy

Make navigation easy by providing a straightforward presentation of the overall work area and the mechanisms for moving through it. Moving easily and quickly within the work area gives the user a sense of mastery over the application and their work. For example, ScrollBars are an effective way to indicate the position of the current view in relation to an area as a whole. In addition to providing positional feedback, ScrollBars allow the user to move through the area.

Arrange elements on the screen according to their use; an optimal arrangement assists the user's decision-making processes and reduces the possibility of errors. The best approach for arranging screen elements according to use is to involve users in the arrangement process. Present screen objects in an orderly, simple, and uncluttered manner.

Reduce mouse movement to simplify the actions of the user. For example, place secondary DialogBoxes near their parent DialogBox so that when the secondary DialogBox appears, the mouse pointer is over the default PushButton, unless the user needs to see the contents of the original DialogBox. Reducing mouse movement helps make an interface natural because, from the user's point of view, work involves a stream of thoughts, intentions, and tasks (some predefined and some that become apparent during the process) that all relate to some desired outcome or accomplishment. This stream of thought is disrupted when the user has to make unnecessary mouse movements, open and close DialogBoxes, or search for commands.

1.4.2 Provide Natural Shades and Colors

Minimize the contrast between screen objects in order to direct the user's attention. Appropriate use of contrast helps the user distinguish screen objects against the background of a window. Very dark screen objects on a light background, very bright objects on a dark background, and bright colors all command the user's attention. If there are many objects with strong contrast or bright colors on the screen, the user will have difficulty knowing where to look first because all these objects compete equally for attention.

Use color as a redundant aspect of the interface; that is, use it to provide additional differentiation among screen objects. Differentiation also comes from the shape and size of the screen objects. For example, in many parts of the world, stop signs are red octagons. You recognize the stop sign by both its shape and color.

1.5 Keep Interfaces Consistent

The main purpose of this *Style Guide* is to ensure consistency. Consistency is important both among applications and within a single application. Consistency helps the user transfer familiar skills to new situations. The user can apply the knowledge learned from one application to another application, reducing the amount of learning and subsequent recall. Consistency within applications facilitates exploration of new functions. When components work in a manner that is consistent with other components, the user will be less afraid to try new functions. The new functions will seem familiar, comfortable, and appropriate. The guidelines in this *Style Guide* allow you to create applications that are consistent in a diverse market and that help your applications succeed in the marketplace.

Intraapplication consistency means the following:

- Similar components operate similarly and have similar uses.

  For example, because PullDown, Popup, and Option Menus are similar components, their operation and use should be similar. Choosing the proper component is described in Section 4.1. Component interaction is described in the reference section for each component and in Section 4.3.

- The same action should always have the same result.

  For example, pushing the top arrow in a ScrollBar should always move the ScrollBar up. Interaction is described in Section 4.3.

- The function of components should not change based on context.

  For example, clicking a button should always perform the same action. Note that even though the action is the same, the result of the action can depend on context. A button in a file editor can begin editing one of a number of files. The button need not always edit the same file, rather its consistent action is to edit the selected file. Interaction is described in Section 4.3.

- The position of components should not change based on context.

  Components should not generally be added and removed as needed. This makes it difficult to quickly find the desired component. Instead, you should make unneeded components nonfunctional and indicate this by deemphisizing (graying out) their labels. Component layout is described in Section 4.2.

- The position of the mouse pointer should not warp.

  The location of the mouse pointer should be determined by direct manipulation and should not be positioned arbitrarily by the application. Positioning the mouse pointer by the application causes the user to lose track of the pointer. Warping the pointer also causes problems with tablet style pointing devices that rely on absolute pointer positioning. The input model is described in Chapter 2.

Consistency among applications increases the user's sense of mastery. Experience with one application can be readily applied to another application, creating a positive transfer of knowledge. The task at hand, rather than learning a new application, becomes the focus of a computer session. When applications work in a manner that is consistent with other applications, users enjoy a feeling of immediate confidence in their ability to master the new program. Also, they are pleasantly surprised when trying new functions because, although new, the functions seem familiar.

Interapplication consistency means the following:

- Components look familiar.

  This does not mean that components look exactly the same, but that the internal layout of components should be the same. Elements of appearance such as color, size, and thickness of beveled edges are less important to application interoperability. Component design and layout are described in the reference section for each component and in Section 4.4.

- Interaction is familiar.

  When interaction is different among applications, it confuses the user and makes it difficult to concentrate on the task of the application. This applies to the behavior of components, input methods, selection models, and keyboard navigation. Interaction is described in Section 4.3.

- Components are organized in a familiar manner.

  The user needs to be able to quickly find the proper component for each task. Organizing the components according to consistent guidelines helps the user do this. Application layout guidelines are described in Section 4.2.

1.6 Communicate Application Actions to the User

Effective applications let the user know what is happening with the application, but without revealing implementation details. Proper communication between the user and the application increases user satisfaction. There are three guidelines for communicating from the application to the user: provide feedback, anticipate errors, and provide warnings.

1.6.1 Give the User Feedback

Feedback lets users know that the computer has received their input. Give users feedback whenever they have selected a component or Menu item by highlighting the component or Menu item in some way. In addition, if certain operations take more than a few seconds, you should let the user know that the computer is working on that operation by providing a message or by changing the pointer to a working pointer.

1.6.2 Anticipate Errors

Anticipate the errors that are likely to occur. By anticipating errors, you can avoid them in your design, enable the support of recovery attempts, and provide messages informing the user of the proper corrective action. For example, one technique for avoiding excessive error messages is to dim interface components when they cannot be used.

Context-sensitive help aids understanding, reduces errors, and eases recovery efforts. Help information text needs to be clear, concise, and written in everyday language. Help information needs to be readily accessible and just as readily removable.

Many users are most comfortable with learning how to use software applications when they use a natural, trial-and-error method. An undo function supports learning by trial and error by minimizing the cost of errors. An undo function allows the user to retract previous actions, and fosters a spirit of exploration and experimentation that is essential.

1.6.3 Use Explicit Destruction

Explicit destruction means that, when an action has irreversible negative consequences, it should require the user to take an explicit action to perform it. For example, while a worksheet can be saved simply by clicking on a Save PushButton, erasing the worksheet should require clicking on an Erase PushButton and answering a warning question like "`Are you sure you want to erase this worksheet?`" with a button click in the warning DialogBox.

Warnings protect the user from inadvertent destructive operations, yet allow them to remain in control of the application. Warnings also encourage the user to experiment without fear of loss. Operations that can cause a serious or unrecoverable loss of data should warn the user of the consequences and request explicit confirmation.

1.7 Avoid Common Design Pitfalls

The process of achieving good design presents many challenges and potential pitfalls. The following guidelines can help you avoid common pitfalls.

- Pay attention to details.

  The details of an application express the sense of craft that you applied to the application. The details of an elegantly designed interface both pleases users and facilitates their work. For example, aligning the PushButtons of two related and overlapping DialogBoxes makes it easier for the user to activate new settings in an apparently seamless operation. Consistent capitalization of Menu items and DialogBox labels is a design detail that reduces textual distractions for the user.

- Do not finish prematurely.

  A common design pitfall is assuming too early that a design is complete. This tendency is aggravated by schedule pressures and difficulty in pinpointing the inadequacies of a design. While it is important to begin designing early, it is also important to allow for redesigning for as long as possible. The first design of an application is not a solution but a fresh perspective from which to view interface design problems.

- Design iteratively.

  Interface design is best done iteratively. The development cycle of implementation, feedback, evaluation, and change avoids errors by allowing for early recognition and correction of unproductive designs.

- Start with a fresh perspective.

  Avoid the temptation to convert existing software by simply translating it to a new style of interface. Because direct manipulation changes the way the user works, a simple one-to-one translation is unlikely to be successful. Command line applications that are converted to direct manipulation need to be extensively reconsidered and revised. The structure of the function hierarchy and presentation needs to be completely redesigned.

- Hide implementation details.

  User interfaces need to hide the underlying software, and present a consistent interface to the user. A good user interface does not allow implementation details of the application to show through; it frees the user from focusing on the mechanics of an application.

Chapter 2

Input and Navigation Models

Consistent models increase the user's sense of control of a system. By implementing consistent models across systems and applications, you encourage that sense of control. This chapter describes the models OSF/Motif uses to move among windows and components and to interact with the components.

- The keyboard focus model, which determines which component on the screen receives keyboard events.
- The input device model, which describes how different input devices, such as the keyboard and the mouse, interact with applications.
- The navigation model, which determines how the keyboard focus is moved among components.

The activation and selection models, also important for system and application consistency, are described in Chapter 3.

2.1 The Keyboard Focus Model

A typical workspace can contain many windows. Each window expects to receive input from the keyboard, the mouse, or both. The window that receives keyboard events has the input focus. Indeed, when keyboard input is directed to a window, it is actually received by some control within the window. The **keyboard focus** model determines which window in the workspace and which component within that window gets each keyboard input. The keyboard focus may also be referred to as the input focus.

The window with the keyboard focus must be highlighted in some way, usually by a change in shade or color to the window border. The component with the keyboard focus must be highlighted by a location cursor. The location cursor is usually a box drawn around the component with the keyboard focus, but can be indicated in a number of ways, like showing or blinking a text insertion cursor, outlining an element of a list, or changing the background shading of a drawing area. Location cursors are described in detail in Section 2.3.2.1.

In order to avoid conflicts, the window manager must allow only one window to have the keyboard focus at a time. Each application must allow only one component at a time to have the keyboard focus within the window that has the keyboard focus.

The keyboard focus model is defined by a focus policy. A focus policy is a specific mechanism for moving the focus among windows and components. This section only attempts to describe the focus policies; however, their impact on window managers, applications, and components is described in detail where it applies. It is sufficient at this point to note that window managers, applications, and new components must support both explicit and implicit focus policies. Chapter 5 describes in detail how to move the focus among windows using explicit and implicit focus policies.

2.1.1 Implicit Focus

In the **implicit focus** policy, the keyboard focus moves to the window into which a user moves the mouse pointer. No explicit action is performed to set the keyboard focus in the implicit focus model. Keyboard events are sent to the window that the mouse pointer is in, more specifically to the component that the mouse pointer is in. In implicit mode, the keyboard focus tracks the mouse pointer. Because of this, there is no way to move the keyboard focus from the keyboard using implicit mode. Implicit focus is sometimes referred to as pointer, track pointer, or track listener policy, or as being real-estate driven. In this focus policy, the location cursor for keyboard events does not need to be shown except in components like Text, in which keyboard input is common; however, the application can show it.

The mouse pointer always behaves as if the focus policy is implicit. That is, mouse events always go to the window and component that the mouse pointer is in.

2.1.2 Explicit Focus

The **explicit focus** policy requires the user to explicitly select which window receives the keyboard focus. Mouse events are sent to the component that the pointer is over, so the mouse pointer must * always behave as if the focus policy is implicit, even when the * focus policy is explicit. In explicit focus mode, a user moves the keyboard focus to a window by pressing **BSelect** while the mouse pointer is over the window. Simply moving the mouse pointer over a window does not give the window the keyboard focus. Because of this, explicit mode is often called click-to-type. In explicit mode, a user moves the keyboard focus to a specific component within a window by pressing **BSelect** over the component. Pressing **BSelect** * must not move focus to a component that is not traversable or does * not accept input. Pressing **BSelect** in a component that is only used to change the visible portion of another component, such as a ScrollBar or Sash, can either give that component focus or not. In * this focus policy, the location of keyboard focus must be shown by a * location cursor.

In explicit mode, focus can also be moved among windows by the keyboard using the **KNextFamilyWindow**, **KNextWindow**, **KPrevWindow**, and **KPrevFamilyWindow** virtual keys. These keys are usually bound to **MAlt <F6>**, **MAlt KTab**, **MAlt KBackTab**, and **MAlt MShift <F6>**, respectively, as shown in the table at the end of this section.

In explicit mode, keyboard focus moves explicitly among the components in a window as well as among the windows. Moving the keyboard focus among components in a window using the keyboard is called **component navigation**. Keyboard events go to the component in the window with the keyboard focus. Keyboard focus is moved among components using **KNextField**, **KPrevField**, **KMenuBar**, **KMenu**, **KDown**, **KLeft**, **KRight**, and **KUp**. These keys are usually bound to **KTab**, **KBackTab**, **<F10>**, **<F4>**, or **<Menu>**, and the arrow keys as shown in the table at the end of this section. Internal window navigation is described in detail later in this chapter.

If the focus policy is implicit, an implementation can still enable keyboard-based operations that explicitly move the location cursor. * It must at least implement the special cases for Menu traversal as * described later in this chapter. **KMenuBar** moves the cursor to the MenuBar. **KMenu** pops up a Popup Menu.

When an implicit focus policy is in use within a window and a * keyboard operation moves the cursor to another component, the * application must temporarily switch to an explicit focus policy. It * must revert to an implicit focus policy when: *

- A mouse button is pressed. *
- A keyboard operation moves the cursor out of a Menu system. *
- A button, other than a CascadeButton, is activated. *
- **KCancel** is typed, except when **KCancel** leaves the location * cursor in a Menu system. **KCancel** is usually bound to * **<Cancel>** or **<Escape>**. *
- A modal operation, such as keyboard-driven dragging, is * completed. *

The following table lists the keyboard navigation bindings.

Table 2-1. Keyboard Navigation Bindings

Virtual Key	Default Bindings
KTab	<Tab>
KBackTab	MShift <Tab>
KNextFamilyWindow	MAlt KTab
KPrevFamilyWindow	MAlt KBackTab
KNextWindow	MAlt <F6>
KPrevWindow	MAlt MShift <F6>
KNextField	KTab
	MCtrl KTab
KPrevField	KBackTab
	MCtrl KBackTab
KMenuBar	<F10>
KMenu	<Menu> or <F4>
KDown	<↓>
KLeft	<←>
KRight	<→>
KUp	<↑>
KCancel	<Cancel> or <Escape>

2.2 The Input Device Model

User interface applications can take input from both pointing devices and keyboards. For whatever reason a user may prefer to interact with an application by using either a keyboard or a pointing device. Some users may even be restricted to using only a keyboard. Because of these preferences and possible restrictions, all features of an application must be available from the keyboard. *

Design your application so that the user can control it using a pointing device, the keyboard, or both. Although you may decide to make the pointing device the primary means of control, you must provide the user with keyboard methods to control the application. Designing applications for dual accessibility enables users to choose the input device they find best suited to their particular work situation and personal preferences. *

2.2.1 Pointing Devices

A pointing device is a tool that lets the user move a pointer around on the screen. A pointing device also has some means of activating the object under the pointer. A pointing device allows the user to directly manipulate screen objects. Rather than entering a keyboard command for each action, a user can point to an object and directly manipulate it using the pointing device.

The most typical pointing device is a mouse, although a graphics tablet, track ball, joystick, and other tools also work as pointing devices. Throughout this guide we use the term mouse to refer to all pointing devices. You can use any pointing device in place of a mouse.

With a mouse, objects on the screen can be manipulated by combining the mouse pointer with the mouse buttons. This guide assumes that a mouse, or any pointing device, has 3-buttons:

BSelect Used for selection, activation, and setting the location cursor. This button must be the leftmost button, except for left-handed users where it can be the rightmost button. *

BDrag Used for moving and copying elements. This button must be the middle mouse button. *

BCustom Used for additional user interactions, including popping up Menus and special dialogs, and for application-specific interactions. This button must be the rightmost button, except for left-handed users where it can be the leftmost button. This guide also mentions the **BMenu** button, which must be bound to **BCustom** on a 3-button mouse. **BMenu** is used for popping up Menus. * * * * * * *

If your pointing device only has two buttons, **BSelect** still must be * bound to the leftmost button. There are a number of alternatives to properly binding **BDrag**, **BCustom**, and **BMenu**.

- **BDrag** can be bound to the rightmost button, and **BCustom** and **BMenu** are then bound to chording both buttons. Chording a button means to press, click, or release two or more buttons simultaneously.
- Alternatively, **BDrag** can be bound to the rightmost button, **BCustom** is undefined, and **BMenu** is then bound to **MCtrl MShift BSelect**.
- Finally, **BDrag** can be left undefined, and **BCustom** and **BMenu** are then bound to the rightmost button.

As with a 3-button mouse, the actions of the rightmost and leftmost mouse buttons can be switched for left-handed users.

If your mouse has only one button, **BSelect** must be bound to it, * **BDrag** and **BCustom** must be left undefined, and **BMenu** must be * bound to **MCtrl MShift BSelect**.

If your mouse has more than three buttons, the first three correspond to those on a 3-button mouse.

Users can combine mouse button press and release actions along with mouse movements to activate elements of applications and components. Various mouse button actions are described as follows, along with this *Style Guide's* usage conventions.

Press — Indicates pressing a mouse button without releasing it. This mouse action is often used to select an object for action. This guide uses **BSelect Press** to indicate a press on the first mouse button.

Release — Indicates releasing a mouse button after pressing it. This mouse action is often used to perform the action initiated by a press or a drag. This guide uses **BSelect Release** to indicate a release on the first mouse button.

Click — Indicates pressing and releasing a mouse button without moving the pointer. This mouse action is used to select an object or perform an action. The

activation action actually occurs on the release of the mouse button. This guide uses **BSelect Click** to indicate a click on the first mouse button.

Drag Indicates pressing a mouse button without releasing it and then moving the position of the pointer. The action of a drag operation tracks the position of the mouse pointer. The drag action ends with a release action. This mouse action is commonly used to select a range of objects, or move a selected object. This guide uses **BSelect Drag** to indicate a drag using the first mouse button.

MultiClick Indicates a number of clicks in quick succession. MultiClicks are often referred to by the actual number of clicks, as in double-click or triple-click. A double-click is often used to perform the default action of an object. This guide uses **BSelect Click 2** to indicate a double-click on the first mouse button, **BSelect Click 3** to indicate a triple-click, and so on. This guide uses **BSelect Click 2+** to indicate two or more clicks on the first mouse button.

MultiPress Indicates a number of clicks in quick succession without a final release. As in MultiClick, MultiPresses are often referred to by the actual number of presses. This guide uses **BSelect Press 2** and the variations mentioned in MultiClick to indicate MultiPress actions on the first mouse button.

MultiDrag Indicates a MultiPress action followed by moving the position of the pointer. The action of a MulitDrag operation tracks the position of the mouse pointer. The MultiDrag action ends with a release action. As in MultiClick, MultiDrags are often referred to by the actual number of presses. This guide uses **BSelect Drag 2** and the variations mentioned in MultiClick to indicate MultiDrag actions using the first mouse button.

2.2.2 Pointer Shapes

The shape of the mouse pointer provides the user with an important visual cue, indicating the functionality of the area in which the mouse pointer is currently located. You should not create new mouse pointer shapes for functions that already have mouse pointer shapes associated with them, except for localization, but you can create new mouse pointer shapes for functions not already associated with a pointer shape. You must not use a predefined * shape to symbolize a function that it was not designed to represent.

A list of the defined pointer shapes follows, along with their usage and hotspots. A pointer's hotspot is the actual position on the pointer that tracks the movements of the mouse. As the pointer * changes from one shape to another, the location of the hotspot must * not move on the screen. The hotspot is the precise location on the pointer where mouse actions occur.

The upper-left pointing arrow pointer is a general-purpose pointer. It is used in most window areas for single-object selection and activation. The hotspot for the arrow pointer should be in the point of the arrow.

You can optionally use an I-beam pointer in any Text component. It is used to change the location of the text insertion cursor and to perform actions on text. If the I-beam is used, it can be hidden during the time between any keyboard action and a mouse movement. This helps the user distinguish the I-beam pointer from the text insertion cursor, which can also be an I-beam. The hotspot for the I-beam pointer should be on the vertical bar of the I-beam about one-third up from the bottom.

The X pointer can indicate when the pointer is outside of any application area. The hotspot for the X pointer should be where the lines intersect.

The resize pointers indicate positions for area resize, and they remain during a resize operation. The direction of the arrow in the pointer indicates the direction of increasing size. The horizontal and vertical pointers

indicate resize in either the horizontal or vertical direction. The diagonal pointers indicate resize in both the horizontal and vertical directions simultaneously. The hotspot for the resizing pointers should be on the elbow or the line at the position pointed to by the arrow.

The hourglass pointer, a working pointer, indicates that an action is in progress in the area, and that the pointer has no effect in the area. While the hourglass pointer is active, all mouse button and keyboard events are ignored in the area. The hotspot for the hourglass pointer should be located at the center of the hourglass, though it should not be used for activation. The hourglass pointer can be used interchangeably with the watch pointer.

The watch pointer, a working pointer, indicates that an action is in progress in the area, and that the pointer has no effect in the area. While the watch pointer is active, all mouse button and keyboard events are ignored in the area. The hotspot for the watch pointer should be located at the top of the watch, though it should not be used for activation. The watch pointer can be used interchangeably with the hourglass pointer.

The 4-directional arrow pointer indicates a move operation in progress, or a resize operation before the resize direction has been determined. During a move operation, the object, or an outline of the object should move to track the location of the pointer. During a resize operation, the pointer is used to indicate a direction for resizing. The 4-directional arrow pointer should change to the appropriate resize arrow when the resize direction is determined, either by crossing an object boundary with the pointer or by pressing a keyboard direction key. The hotspot for the 4-directional arrow pointer should be at the spot where the arrows intersect.

The sighting pointer is used to make fine position selections. For example, in a drawing program it may be used to indicate a pixel to fill or the connecting points of lines. The hotspot for the sighting pointer should be at the spot where the lines intersect.

The caution pointer is used to indicate that action is expected in another area before input can be given to the current area, and that the pointer has no effect in the area. While the caution pointer is active, all mouse button and keyboard events are ignored in the area. The hotspot for the caution pointer should be located at the center of the caution symbol, though it should not be used for activation.

The question pointer is used to request an input position or component from the user. This is often used to input an object for interactive help. The user requests interactive help, then the question pointer is displayed to allow the user to indicate what position or component help is requested for. The hotspot for the question pointer should be at the bottom of the question mark.

You can optionally use an arrow pointing to the upper right corner to indicate a pending Menu action. This shape indicates that a Menu is popped up or pulled down and waiting for a Menu item to be activated or the Menu to be removed. The hotspot for this arrow pointer should be in the point of the arrow.

If you need to design a new mouse pointer shape, keep the following in mind:

- The shape of the pointer should give a hint to its purpose.
- The hotspot should be easy to locate, and obvious.
- The shape should be easy to see.
- The shape should not create visual clutter.

The details of how the pointer tracks the motion of the mouse are not handled at the level of the application, but they are worth noting here. There are two concepts that define how the pointer tracks the motion of the mouse: gain and acceleration.

Gain refers to the ratio of the distance the pointer moves to the distance the mouse moves. If the gain is increased, the mouse pointer moves farther for a given mouse movement. The gain should not change across the environment. Instead, if your application requires finer motion, it should include a zoom feature. A zoom feature changes the relative size of an area of your application. Zooming in on an area allows the user to make finer adjustments than normal. Zooming out from an area allows the user to move through the application more quickly.

Acceleration is a temporary change in the gain. It is commonly used to change the gain, based on the speed of the mouse movements. The acceleration could be set so that if the mouse moves slowly, the gain is reduced to allow for finer adjustments of pointer position, or if the mouse moves quickly, the gain is increased to allow quicker screen movement. This can be very confusing to new users. Gain and acceleration are handled on a global scale in the system, outside of the application. Applications should not change the gain and acceleration characteristics of mouse movement.

2.2.3 Warp Pointer Only If Explicitly Enabled

The pointer position is intended only as input to applications. It is not intended as an output mechanism for applications. Changing the location of the mouse pointer is known as warping the pointer. * Your application must not warp the pointer, unless you provide the * user with a means of disabling this behavior.

Warping the pointer is confusing to users, and reduces their sense of control. Also, warping the pointer can cause problems for users of absolute location pointing devices (like graphics tablets). Graphics tablets map pointer device locations to absolute screen locations, so if the pointer is warped, the pointer loses synchronization with the pointing device, making some screen locations impossible to reach.

Note that for these types of devices, accelerating the pointer has the same effect as warping the pointer, but this can be handled by the pointer driver software.

2.3 The Navigation Model

Regardless of whether they use a mouse, a keyboard, or both, users need to move the mouse pointer or the location cursor to new positions. That is, they will need to navigate around the workspace. The model is simple for mouse users, and more complicated for keyboard users. This section describes the general navigation model for mouse and keyboard users, describes the more complicated Menu navigation model, and then describes navigation within scrollable components.

2.3.1 Mouse-Based Navigation

In mouse-based navigation, the pointer tracks the motions of the pointing devices. This is described earlier in this chapter, and is known as the implicit focus policy. The mouse must always behave as if the focus policy is implicit. Components that are used to adjust the size and location of other elements need not move the location cursor when they receive the keyboard focus. Instead, after the mouse has acted on the component, the focus should remain on the component that previously had it.

The only exception to the simple model of pointer navigation is a Menu system because Menus are not available on the screen until activated. Activating a Menu causes it to be shown on the screen. There are three types of Menus: PullDown Menus, Popup Menus, and Option Menus. The MenuBar is also a special kind of Menu. A **Menu system** consists of all the Menus cascading from a single CascadeButton, OptionButton, or Popup Menu. The **MenuBar system** consists of the MenuBar and all of its associated Cascade Menus.

A PullDown Menu is generally activated by pressing **BSelect** on a CascadeButton, which displays the PullDown Menu. A Popup

Menu is generally activated by pressing **BMenu** in an area with a Popup Menu, which displays the Popup Menu. An Option Menu is generally activated by pressing **BSelect** on an OptionButton, which displays the Option Menu. A PullDown Menu pulled down from a CascadeButton within a PullDown, Popup, or Option Menu is called a CascadeMenu. A MenuBar is generally activated by moving the input focus to the MenuBar. Since a MenuBar is always visible, activating the MenuBar does not change its appearance. Navigating to a Menu is equivalent to activating a Menu. **BSelect Press** with the pointer on a Menu must activate the Menu in a spring-loaded manner. Spring loaded means that the Menu is removed when the button is released, except on a CascadeButton. **BMenu Press** with the pointer in an area with a Popup Menu must activate the Menu in a spring-loaded manner. Once a Popup Menu is posted, **BMenu** must behave just as **BSelect** is described for any Menu system. **BSelect** must also be available from within a posted Popup Menu system, and must behave just as in any Menu system. **BSelect Release** within a spring-loaded Menu system must activate the button under the pointer at the time of the release. Note that if the release is on a CascadeButton, the associated Cascade Menu must be posted.

Whenever any Menu is popped up or pulled down, the location cursor must be placed on the Menu's default entry, or on the first entry in the Menu if there is no default entry. Support for default entries in Menus is optional.

A spring-loaded Menu must be removed when the mouse button that activated it is released. While a spring-loaded Menu system is popped up or pulled down, moving the pointer within the Menu system must move the location cursor to track the pointer. If the pointer rests on a CascadeButton, the Menu must be pulled down and must also become traversable. The Menu must be removed once the pointer moves to a Menu item outside of the Menu or its CascadeButton. If the Menu system is part of a MenuBar, moving the pointer to any other element on the MenuBar must unpost the current Menu system and post the PullDown Menu associated with the new element. Releasing the button must activate any component in the Menu system, including a CascadeButton. Note that releasing the mouse button in a CascadeButton, even a CascadeButton in the MenuBar, causes the associated Menu to remain posted.

BSelect Release with the pointer on a CascadeButton or OptionButton must activate the associated Menu in a posted manner. **BMenu Release** with the pointer in an area with a Popup Menu must activate the Menu in a posted manner. A posted Menu remains visible until explicitly unposted. The keyboard focus model is forced to explicit, and the Menu system can be traversed using the keyboard as described later in this section. A posted Menu must not be unposted until one of the following events occurs:

- A mouse button is pressed.
- A keyboard operation moves the cursor out of a Menu system.
- A keyboard operation moves the cursor to a parent of the posted Menu.
- A Menu item, other than a CascadeButton, is activated.
- **KCancel** is typed from the top of the Menu system.

If a mouse button is pressed outside of the Menu to unpost the Menu, the button press and its release can also effect the underlying component, otherwise both the press and the release must have no additional effect. Either both the press and release must effect the underlying component, or neither the press nor the release must effect the underlying component.

Once a PullDown or Option Menu is posted, **BSelect Press** in the Menu system must cause the Menu to behave as a spring-loaded Menu. Once a Popup Menu is posted, **BMenu Press** in the Menu system must cause the Menu to behave as a spring-loaded Menu. Because of this, the pointer driven activation behavior of spring-loaded Menus fully describes pointer driven activation behavior in Menus.

2.3.2 Keyboard-Based Navigation

The navigation model for keyboard users is more complicated than the pointer navigation model. As described earlier, keyboard navigation is only required when the focus policy is explicit. Since window managers, applications, and components must support the

explicit focus policy, they must also support keyboard navigation. The keyboard navigation model is composed of the following:

- A focus identifier (the location cursor)
- Navigating among windows
- Navigating within windows

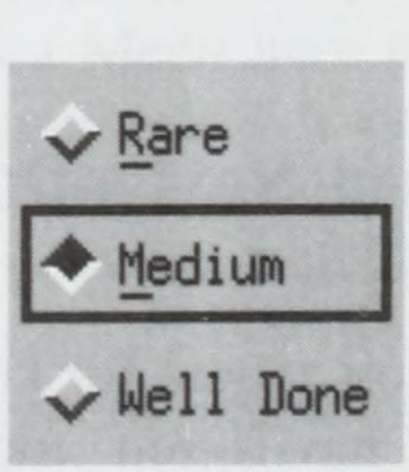
2.3.2.1 Location Cursor

The window with the focus is identified by highlighting the window border as described in Chapter 5. Within the window with the focus, the component with the keyboard focus is identified by the location cursor. The term location cursor refers to any visual element that indicates the location where keyboard events are sent. The location cursor is not strictly a cursor. The name cursor simply refers to its use as indicating the location of input.

The location cursor is shown in a number of ways, depending on the type of component with the keyboard focus. Possible location cursor types and their uses are described in the following text. When designing new components, you should try to use one of the existing styles of location cursor, but you can create your own if it is appropriate.

Box The box cursor must be the default location cursor. It is shown as a box drawn around the object. Figure 2-1 shows the first ToggleButton in a group with a box style location cursor.

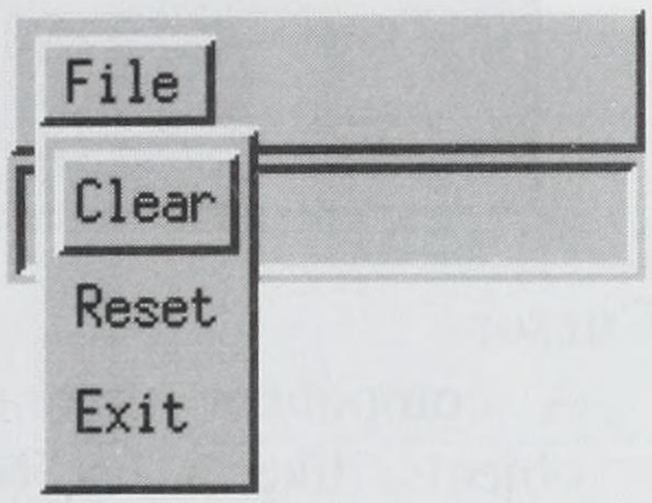
Figure 2-1. A Box Style Location Cursor



Outline Highlight

This cursor style is similar to the box style cursor. The outline is shown of a component whose outline is not normally shown. This form of location cursor is commonly used within Menu systems to show the Menu item with the location cursor. Figure 2-2 shows a Menu with an outline highlight style location cursor.

Figure 2-2. An Outline Highlight Style Location Cursor



Text Cursor

In Text components, the text cursor acts as the location cursor to indicate that the Text component has the keyboard focus. Figure 2-3 shows possible text cursor shapes.

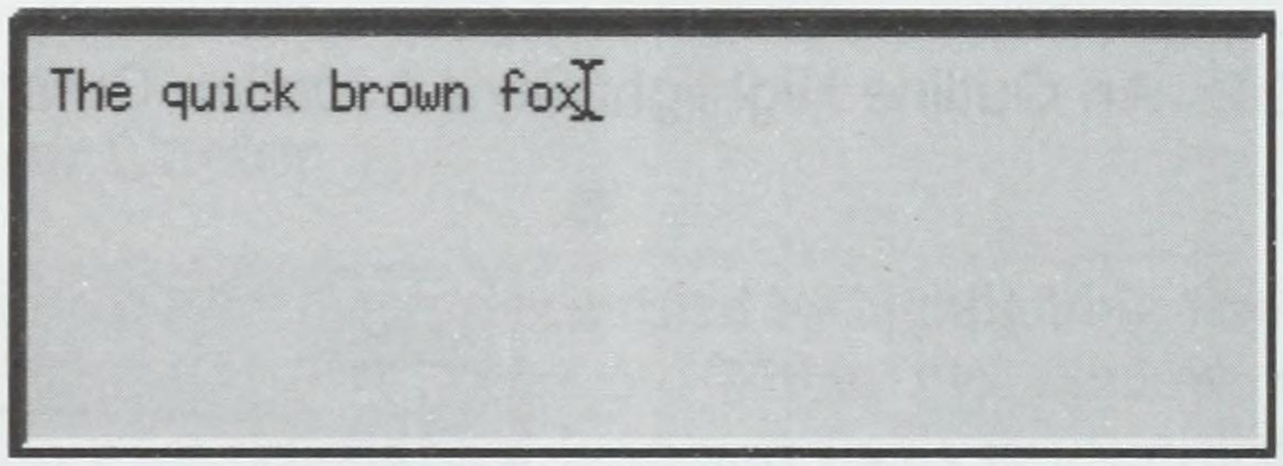
Figure 2-3. Text Cursor Shapes

	Insertion	Overstrike
Active	|	▮ or _
Inactive	⋮ or ^	▒ or ...

A text cursor must be shown differently when the the Text has the keyboard focus. This can be done by graying and emboldening, making it static and blinking, or shown and hidden. If the text cursor is hidden when the Text component does not have the focus, then when it is traversed to, the text cursor must appear in the same location that it was in the last time it was traversed out of. The initial position of the text cursor must be at the beginning of the text. A caret is also used to show the destination cursor in Text components. A Text component

can optionally include an outline highlight style location cursor to reinforce the location of the keyboard focus. Figure 2-4 shows a Text component with the keyboard focus.

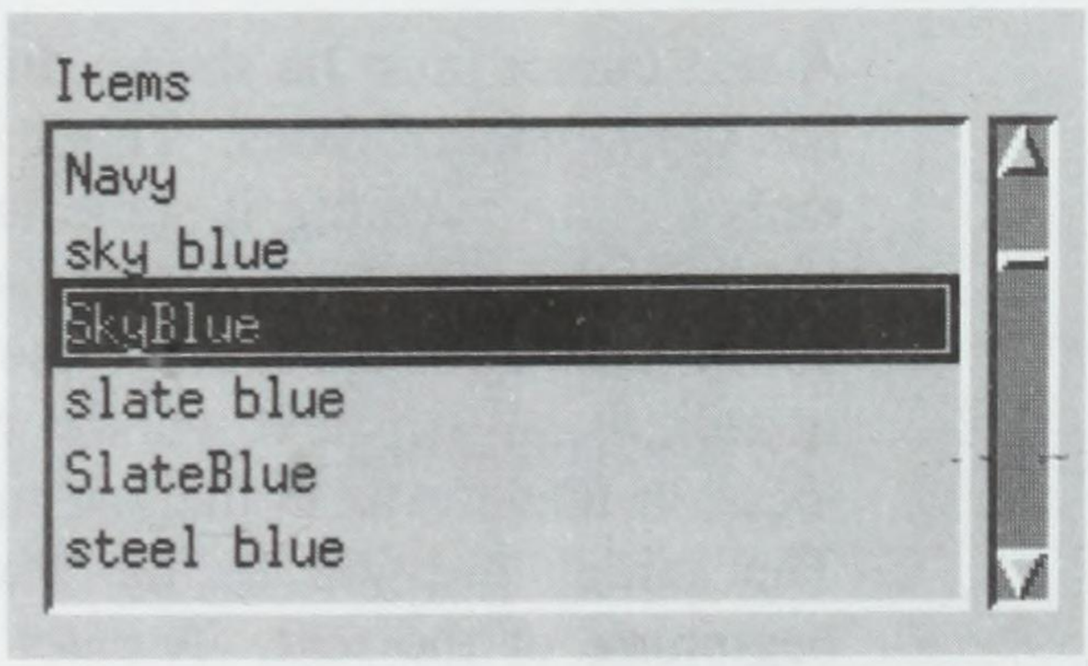
Figure 2-4. A Text Cursor Style Location Cursor



Item Cursor

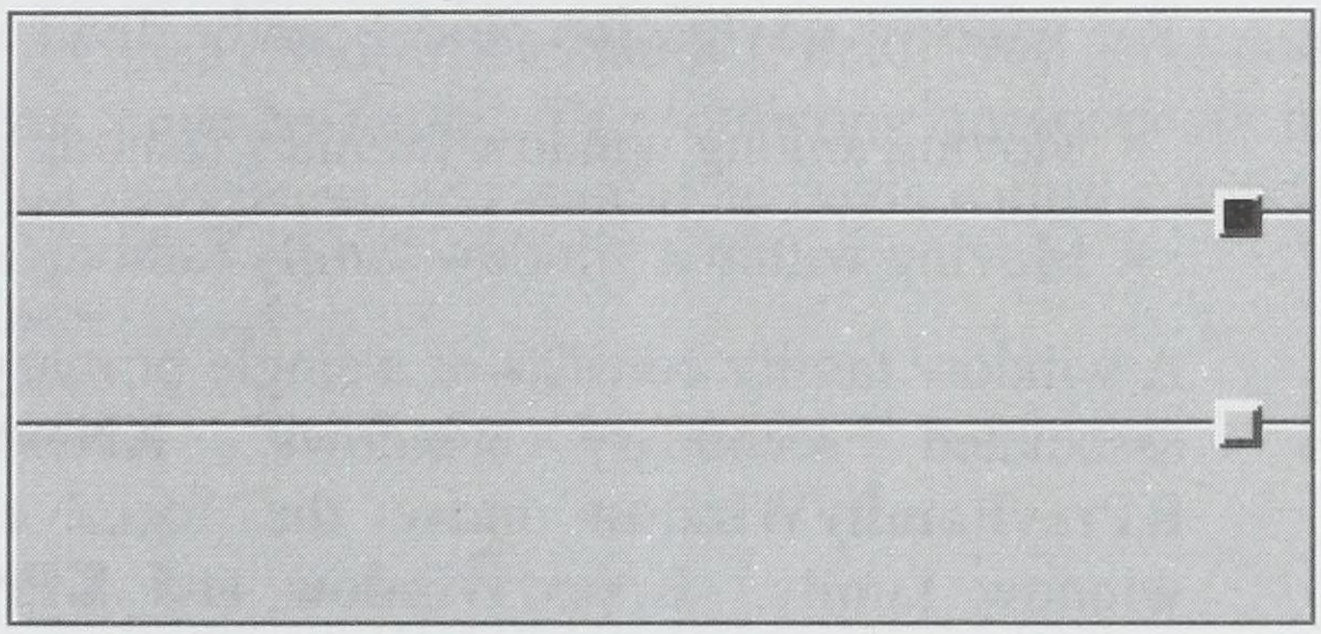
A component that manages groups of noncomponent objects, like a list box, should indicate that it has the keyboard focus with an item cursor style location cursor. An item cursor highlights a single object, or group of objects, that the component acts on. In the case of a List, the item cursor should be a box around an item. In the case of a Drawing area, it could be a box with resize handles around a drawn object. Components that use an item cursor to indicate keyboard focus can optionally include an outline highlight style location cursor to reinforce the location of the keyboard focus. Figure 2-5 shows a List component with the keyboard focus.

Figure 2-5. An Item Cursor Style Location Cursor



Fill Some very small components, like a Sash for resizing Panes, should indicate the keyboard focus by filling. * Where this is the case, there must be no other meaning * associated with the filled state. Filling avoids the problem caused by other styles of location cursor that make the small component too large. Figure 2-6 shows the first of two Sashes with the keyboard focus.

Figure 2-6. A Fill Style Location Cursor



Components must be designed and positioned within applications so * that adding and removing the component's location cursor does not * change the amount of space that the component takes up on the * screen. That is, there is always room for the location cursor.

2.3.2.2 Window Navigation

A typical environment has several applications in operation simultaneously. Each application typically has a main or primary window that displays data and in which the user carries on primary interaction with the application. Applications can have additional windows to communicate context-specific interactions with the user of the application. These additional windows are called secondary windows, or transient windows. DialogBoxes are often used to create secondary windows.

The **window navigation** model can be divided into two levels:

- Moving among window families (among primary windows)
- Moving within a window family (among secondary windows)

A window family consists of a single primary window and all of its associated secondary windows. **KNextFamilyWindow** and **KPrevFamilyWindow** move the focus among windows in a window family. **KNextWindow** and **KPrevWindow** move the focus among window families. Window navigation is described in more detail in Chapter 5. The following table summarizes the common window navigation bindings.

Table 2-2. Window Navigation Bindings

Virtual Key	Default Bindings
KTab	<Tab>
KBackTab	MShift <Tab>
KNextFamilyWindow	MAlt <F6>
KPrevFamilyWindow	MShift MAlt <F6>
KNextWindow	MAlt KTab
KPrevWindow	MAlt KBackTab

2.3.2.3 Component Navigation

Component navigation moves the location cursor, and therefore the keyboard focus, among components within a window. Some components, like Text, Menus, and Lists, have additional keyboard methods of traversal, which are described in the behavior descriptions of those components. The model for component navigation organizes components into traversable groups called fields or tab groups. Components that have their own **internal navigation**, such as a List, must be a field. Groups must be fields, unless the group has no controls in it. When a component is used as a Pane of a PanedWindow it must be a field. Top level components in a window must be fields. The following components have their own internal navigation, and should be fields within a window.

- Canvases
- Text
- Lists
- Sashes
- ScrollBars
- Scales

Single line Text components can be part of another field, rather than a field of their own.

Menu systems, including the MenuBar, follow a different traversal model that is described later in this section.

Moving the location cursor among field components and fields in a window must be done with **KNextField** and **KPrevField**. Fields are often called tab groups because **KNextField** and **KPrevField** are usually bound to **<Tab>** and **<BackTab>** as shown in the following table. Some controls, such as Text, use **<Tab>** and **<BackTab>** internally. These controls must use the **<Ctrl>** modified bindings for **KNextField** and **KPrevField**.

Table 2-3. Field Navigation Bindings

Virtual Key	Default Bindings
KTab	<Tab>
KBackTab	MShift <Tab>
KNextField	KTab
	MCtrl KTab
KPrevField	KBackTab
	MCtrl KBackTab

KNextField must move the location cursor forward through fields according to the following rules:

- If the next field is a field control, **KNextField** must move the location cursor to it.
- If the next field is a field group, **KNextField** must move the location cursor to the first traversable component within the field group, unless the field contains a default component.
- If the field contains a traversable default component, **KNextField** must move the location cursor to the default component instead of the first component, unless the application arranges for the cursor to be placed on some other component in the field.

Application developers should generally make a default component the first component in a field. Section 3.3.6 describes the use of default components in detail. If the next field contains no traversable components, the components are fields, or the components are inside nested fields, **KNextField** must skip the field.

KPrevField must move the location cursor backward through fields using the same rules as **KNextField**.

Developers should follow these general rules for field navigation:

- Fields should be traversed from the upper-left to the lower-right in a left-to-right direction language environment. In a right-to-left direction language environment, fields should be traversed from upper-right to the lower-left.
- Field navigation must wrap between the first and last field. *
- If one field is the child of another field, **KNextField** should traverse the parent field before the child.
- In the case of a PanedWindow, each Sash and Pane must be a * field, and must be traversed in top-to-bottom order. Any fields * within a Pane must be traversed before the next Sash.
- If the field's default component is in a nested field, these rules can be relaxed to make sure that the field and all of its descendents are traversed by **KNextField** before traversing to another field.

When using an explicit keyboard focus policy, and a window gets the focus, the location cursor should be placed in the last component in that window to have the location cursor. If that component is no longer traversable, or if the window has never had the location cursor before, the cursor should be placed in the first traversable component in the window. If the window is a DialogBox, the location cursor should be placed on the default component in the DialogBox, though an application can place the cursor on any specific component in the DialogBox. If the default component is not traversable, or if there is no default component, the location cursor should be placed as for any other window.

When using an explicit keyboard focus policy, and a field gets the focus, the location cursor should be placed in the first traversable component in the field. If the field is a DialogBox nested in a window (that is, a field with a default control), the location cursor should be placed on the default component in the DialogBox, though an application can place the cursor on any specific component in the DialogBox. If the default component is not traversable, or if there is no default component, the location cursor should be placed as for any other field.

KDown, **KLeft**, **KRight**, and **KUp** must move the location cursor * among components in a field group, or among elements of a field *

control. In a graphics-like field, these keys can move the location cursor by an element or by a pixel. These keys modified with **MCtrl** can move the cursor in a large increment. What an increment and a large increment is depends on the type of the component. For example, in a scale an increment could be a unit of the scale, and a large increment could be a number of units, or the distance between major tick marks on the scale. **KDown**, **KLeft**, **KRight**, and **KUp** usually are bound to the arrow keys as shown in the following table.

Table 2-4. Component Navigation Bindings

Virtual Key	Default Bindings
KDown	**<↓>**
KLeft	**<←>**
KRight	**<→>**
KUp	**<↑>**

Navigation within a field must behave as described in the following text. Note that **KDown** and **KRight** do not need to traverse components or elements in the same order. *

KDown — In a left-to-right direction language environment must move the location cursor through all traversable components or elements in the field, starting at the upper-left and ending at the lower-right then wrapping back up to the upper-left. In a right-to-left direction language environment the location cursor can move, starting at the upper-right and moving to the lower-left. *

KUp — Must move the location cursor through all traversable components or elements in the opposite direction of **KDown**. *

KRight — Must move the location cursor through all traversable components or elements in the field, starting at the upper-left and ending at the lower-right then wrapping back up to the upper-left. *

KLeft — Must move the location cursor through all traversable components or elements in the opposite direction of **KRight**. *

Separators and Labels should not be traversable. Labels can be traversable if the system supports quick transfer of the Label text or graphic. The ScrollBars of ScrolledWindow need not be traversable.

Field controls should also support the following behavior. Component navigation in field groups can also support this behavior.

KBeginLine In a left-to-right direction language environment, this action should move the location cursor to the leftmost edge of the data.

KEndLine In a left-to-right direction language environment, this action should move the location cursor to the rightmost edge of the data.

KBeginData Should move the location cursor to the first element or the topmost or topmost left edge of the data.

KEndData Should move the location cursor to the last element or the bottommost or bottommost right edge of the data.

These keys are usually bound as shown in the following table.

Table 2-5. Component Navigation Bindings

Virtual Key	Default Bindings
KBeginData	MCtrl KBeginLine
KEndData	MCtrl KEndLine
KBeginLine	<Begin> or <Home>
KEndLine	<End>

2.3.3 Menu Traversal

The Menu traversal model is different from the field traversal model. This is so that Menus are traversable even when the focus policy is implicit. If a Menu is traversed to while the focus policy *

in the application is implicit, the focus policy must temporarily change to explicit. The focus policy must revert to implicit whenever any one of the following occurs:

- A mouse button is pressed.
- A keyboard operation moves the cursor out of a Menu system.
- A Menu item, other than a CascadeButton, is activated.
- **KCancel** is typed from the top of the Menu system.

Traversing to a Menu system is the same as activating the Menu system. If the MenuBar is inactive, **KMenuBar** must traverse to, or activate, the MenuBar system. The location cursor must be placed on the first traversable Menu in the MenuBar. If there are no traversable Menus in the MenuBar, **KMenuBar** must do nothing. If the MenuBar system is active, **KMenuBar** must unpost all Menus in the MenuBar system and return the location cursor to the component that previously had it.

If the keyboard focus is in an area with an inactive Popup Menu, **KMenu** must post (activate) the Popup Menu. The location cursor must be placed on the first traversable item in the Popup Menu. If there are no traversable items in the Popup Menu, it is up to the system and the application whether to post the Menu or not. If there is an active Popup Menu, **KMenu** must unpost the Popup Menu system and return the location cursor to the component that previously had it.

mS If the keyboard focus is in an OptionButton, **KActivate** or **KSelect** must post, or activate, the Option Menu. The location cursor must be placed on the previously selected item in the Option Menu. If the Option Menu is pulled down for the first time, the location cursor must be placed on the default item in the Menu. If there are no traversable items in the Option Menu, the application should decide whether to post the Menu or not. If there is an active Option Menu, **KActivate** or **KSelect** must select the current item in the Option Menu, unpost the active Option Menu system, and return the location cursor to the OptionButton.

KMenuBar and **KMenu** are usually bound to **<F10>**, and **<F4>** or **<Menu>**, respectively. **KActivate** is usually bound to **<Return>** or **<Enter>**. **KSelect** is usually bound to **<Select>**, **<Space>**, and **MCtrl <Space>**, The following table shows the common Menu traversal bindings.

Table 2-6. Menu Traversal Bindings

Virtual Key	Default Bindings
KMenuBar	<F10>
KMenu	<F4>
KActivate	<Return>
	<Enter>
	MCtrl <Return>
	MCtrl <Enter>
KSelect	<Select>
	<Space>
	MCtrl <Space>

Once a Menu system is posted, the Menu items can be traversed using **KNextMenu**, **KPrevMenu**, **KDown**, **KLeft**, **KRight**, and **KUp**. A posted Menu system behaves somewhat like a field with the addition of traversing among Menus in the system. All Menu keyboard traversal actions wrap between the first and last elements. * The system must support the Menu traversal behavior described in * the following text:

KNextMenu If the component with the keyboard focus is a * CascadeButton, traverse to the default entry in the * associated Cascade Menu, or to the first traversable * entry in the Menu if the Menu has no default. *

Otherwise, if the current Menu has a horizontal * Menu ancestor including the MenuBar, unpost the * current Menu system pulled down from that ancestor * and traverse to the next entry in the horizontal Menu. * If that entry is a CascadeButton, post its associated * PullDown Menu and traverse to the default entry in * the Menu, or to the first traversable entry in the Menu * if the Menu has no default.

KNextMenu is usually bound to **MCtrl <↓>** and **MCtrl <→>**.

KPrevMenu If the current Menu is the child of a vertical or 2-dimensional Menu, traverse to the parent CascadeButton.

Otherwise, if the current Menu is the child of a horizontal Menu or the MenuBar, unpost the current Menu system pulled down from that parent and traverse to the previous entry in the horizontal Menu. If that entry is a CascadeButton, post its associated PullDown Menu and traverse to the default entry in the Menu, or the first traversable entry in the Menu if the Menu has no default.

KPrevMenu is usually bound to **MCtrl** <↑> and **MCtrl** <←>.

KDown If the component is in a vertical Menu or a 2-dimensional Menu, traverse down to the next traversable component. If the component is in a horizontal Menu or a MenuBar, perform the same action as for **KNextMenu**.

KUp If the component is in a vertical Menu or a 2-dimensional Menu, traverse up to the next traversable component. If the component is in a horizontal Menu, perform the same action as for **KPrevMenu**.

KLeft If the component is in a vertical Menu, perform the action of **KPrevMenu**. If the component is in a horizontal Menu, MenuBar, or 2-dimensional Menu, traverse left to the next traversable component.

KRight If the component is in a vertical Menu, perform the same action as for **KNextMenu**. If the component is in a horizontal Menu, MenuBar, or 2-dimensional Menu, traverse right to the next traversable component.

KCancel is used with posted Menus to back out of the Menu hierarchy. Its effect depends upon the location of the cursor:

- In a PullDown Menu, **KCancel** must dismiss the Menu and move the location cursor to the CascadeButton used to pull it down. *

- In a Popup Menu or Option Menu, **KCancel** must dismiss the Menu. If an explicit focus policy is in use, the location cursor must be moved back to the component that previously had it. *

- In a MenuBar, if an explicit focus policy is in use, **KCancel** must move the location cursor to the component that previously had it. *

The following table shows the common internal Menu navigation bindings.

Table 2-7. Menu Navigation Bindings

Virtual Key	Default Bindings
KNextMenu	MCtrl <↓>
	MCtrl <→>
KPrevMenu	MCtrl <↑>
	MCtrl <←>
KDown	<↓>
KLeft	<←>
KRight	<→>
KUp	<↑>
KCancel	<Escape> or <Cancel>

2.3.4 Scrollable Component Navigation

Certain components, such as List and Text, have built-in support for scrolling. However, any component or group of components can be associated with scrolling components or placed inside a ScrolledWindow and made scrollable.

A scrollable component generally has ScrollBars or some other type of scrolling component associated with it. When a component does not have a scrolling component associated with it, it generally

should not be scrollable; however, components whose visible contents alone indicate that additional items exist beyond the bounds of the visible area, like Text, can be scrollable even if they do not have an associated scrolling component.

Any scrollable component must support appropriate navigation and scrolling operations. You must use the page navigation keys **KPageUp**, **KPageDown**, **KPageLeft**, and **KPageRight** for page-at-a-time scrolling within components or groups of components that support scrolling. A page is the portion of data that is visible, not any underlying structure of the data. When scrolling by a page, you must leave at least one unit of overlap between the old and new pages; for example, a line in a Text component.

You should use the directional keys **KUp**, **KDown**, **KLeft**, and **KRight** for either moving the location cursor among its elements, moving it by increments, or for scrolling the visible region by a regular increment. In general, keyboard operations should traverse through the entire scrollable component, not just through the visible portion. If the directional keys only traverse through the visible portion, the ScrollBar (or other scrolling component) that changes the visible portion of the scrollable component must itself be traversable.

In a vertically scrollable component, you should support **KBeginData** and **KEndData** to move to the beginning and end of the scrollable component. In a horizontally scrollable component, you should support **KBeginLine** and **KEndLine** to move to the leftmost and rightmost edge of the scrollable component.

The following table shows the common bindings for the virtual keys used for scrolled region navigation.

Table 2-8. Scrolled Region Navigation Bindings

Virtual Key	Default Bindings
KUp	<↑>
KDown	<↓>
KLeft	<←>
KRight	<→>
KPageUp	<PageUp>
KPageDown	<PageDown>
KPageLeft	MCtrl <PageUp>
KPageRight	MCtrl <PageDown>
KBeginData	MCtrl KBeginLine
KEndData	MCtrl KEndLine
KBeginLine	<Begin> or <Home>
KEndLine	<End>

When a mouse button is pressed initiating a selection operation within a scrollable component, and the pointer is then dragged outside of the scrollable component, the component should scroll toward the pointer. Drag and drop operations should produce similar scrolling behavior within scrollable components that force dragged elements to remain within the component.

When the location cursor is within a scrollable component, scrolling, either by dragging if the cursor is within the scrollable component, by using a ScrollBar, or by using the page navigation keys, may move the cursor out of view; however, if the location cursor is within the scrollable component, any keyboard operation that moves the cursor or that inserts, deletes, or modifies items at the cursor location must scroll the component so that the cursor is visible when the operation is complete. When scrolling using the mouse, the location cursor can be allowed to scroll out of the visible region. When scrolling using the keyboard, the location cursor should be moved so that it remains within the visible region. *

Chapter 3

Selection and Component Activation

The selection model determines how elements are selected from a group of elements. This chapter describes the selection models that are used by the various components of an application. It then describes the component activation model, which determines how to act upon the selected elements.

OSF/Motif compliant applications use an object-action selection model. In an object-action selection model, the user first selects an object, and then performs an action on it. The object-action selection model mimics real life, so it is a familiar process to the user and increases the user's sense of control over applications.

Objects include not only recognizable objects like windows, PushButtons, and List elements, but also component elements that are not always recognized as discrete objects, like individual letters of a text file. Applications can also treat a group of elements as a single element, either in a contiguous range or as a noncontiguous group.

Once users select an object, they can perform an action on it by using the components available in the application, or by using one

of the selection actions, like cut, copy, and paste. Using components to perform actions on a selection is called activation. This chapter describes the following topics:

- Selection models, which determine how elements in a collection are selected for later action.
- Selection actions, which perform actions directly on a selection, such as cut, copy and paste.
- Component activation, which is the means of using components with a mouse or keyboard.

3.1 Selection Models

Selection is used to mark one or more of a group of elements simply for highlighting the elements, or so they can be moved, copied, or otherwise manipulated by the application.

Groups of elements can be organized into the following three types of collections:

- List-like
- Text-like
- Graphics-like

In list-like collections, when the pointer or location cursor is in the collection, the pointer or location cursor is considered to be on an element. The notion of "on" depends upon the manipulation model presented by the control. For example, "on" may mean that the cursor is within the bounds of the element, or it may mean that the cursor is on the boundary of the element, or the element may have handles, and "on" means that the cursor is on one of the handles. In text-like collections, when the pointer or location cursor is in the collection, the pointer or location cursor is considered to be between two elements. The name text-like refers to Text components in insert mode where the insertion cursor is always between two letters. List-like and text-like collections are usually ordered as linear collections. That is, the collections are treated as if the elements were connected as a series of elements in one dimension.

In graphics-like collections, the pointer and location cursor can be either on or between elements. Graphics-like collections are usually ordered as 2-dimensional collections. That is, the ordering of elements in the collection depends on both the horizontal and vertical position of the element within each collection.

OSF/Motif compliant systems must support the five different * selection models. Each model is used where appropriate in applications to limit the type of selections the user can make. For example, a List may limit selection to a single element where the user is only allowed to make one choice. The five selection models are as follows:

Single Selection

Used to select a single element in a collection. Clicking **BSelect** on an element selects it and deselects the previously selected element in the collection. Single selection is described in Section 3.1.1

Browse Selection

Used to allow browsing through single selection collections. Browse selection is also used to select a single element of a collection. Browse selection works just like single selection, but additionally allows the user to browse through the elements by dragging **BSelect** through the elements in the collection. Browse selection highlights each element as it is traversed, and gives the application an opportunity to provide information about each element as it is highlighted. Releasing **BSelect** on an element selects it and deselects the previously selected element. Browse selection is described in Section 3.1.2

Multiple Selection

Used to select or deselect multiple elements of a collection. Clicking **BSelect** on an unselected element adds that element to the current selection. Clicking **BSelect** on a selected element removes that element from the current selection. Multiple selection is described in Section 3.1.3

Range Selection

Used to select a contiguous range of elements in a collection. Clicking **BSelect** on an element selects the single element, and deselects any previous selection. **BSelect Drag** over a range of elements selects all the elements within the range, and deselects any previous selection. Range selection is described in Section 3.1.4

Discontiguous Selection

Used for selecting multiple discontiguous ranges of elements in a collection. Clicking or dragging **BSelect** operates just as for range selection. Discontiguous selection also allows **BToggle** to be used to add or remove selection ranges. Discontiguous selection is described in Section 3.1.5

The choice of the selection model should be made on a collection by collection basis. Some components, like Text, enforce a selection model. Other components, like a List, allow the application or the user to determine the selection model.

Variants of these selection models may be needed for collections that are especially dense, when objects are organized in layers, when 3-dimensional selection is needed, when objects are hierarchicly organized, or in other situations not covered by this *Style Guide*. Selection models for such cases should not deviate unnecessarily from the standard selection models.

In all selection models, the selected element or group of elements should be visually highlighted in some way. Highlighting the selection can be done in the following of ways:

- Using any one of the location cursor mechanisms
- Inverting the colors of the selection
- Drawing a solid or dashed box around the elements of the selection

The selection can also be a null selection. That is, it is possible for no items to be selected in a collection. This commonly occurs in text-like collections where the location cursor is between two elements, or when the user has deselected all the elements.

3.1.1 Mouse-Based Single Selection

The single selection model is the simplest selection model. In the single selection model, when **BSelect** is clicked in a selectable element that is deselected, that element must be selected, and any other selection in the collection must be deselected. The selected element must have the location cursor.

3.1.2 Mouse-Based Browse Selection

The browse selection model is very similar to the single selection model. In browse selection, like single selection, only a single element is selected at a time. In the browse selection model, when **BSelect** is released in a selectable element, that element must be selected, and any other selection in the collection must be deselected. As **BSelect** is dragged through selectable elements, each element under the pointer must be selected, and the previously selected element must be deselected. The selection must remain on the element where **BSelect** is released and the location cursor must be moved there. The browse selection model gives the application an opportunity to provide information about each element as it is highlighted.

3.1.3 Mouse-Based Multiple Selection

Multiple selection allows the user to select or deselect multiple single elements of a collection. Clicking **BSelect** on an unselected element must add that element to the current selection in the collection. Clicking **BSelect** on a selected element must remove that element from the current selection in the collection. Clicking **BSelect** on an element must move the location cursor to that element.

With multiple selection, the user can select any group of elements in a collection, including discontiguous groups, all the elements, or none of the elements. Because multiple selection requires one mouse click for each element selected, although a simple model, it is not well-suited for selecting large groups of elements.

3.1.4 Mouse-Based Range Selection

The range selection model allows the user to select multiple contiguous elements of a collection. The description of this selection model assumes that the drag-and-drop model is not in use in the collection. The drag-and-drop model and its effect on this selection model are described in Section 3.2.1.

In the range selection model, when **BSelect** is pressed, all elements in the collection must be deselected. If **BSelect** is clicked on a selectable element, it must be selected. Pressing **BSelect** must also set an anchor on the element, or at the position where **BSelect** was pressed. The **anchor** is one of the endpoints of the selected range. As **BSelect** is dragged through the collection, every element between the pointer and the anchor must be included in the selection. When **BSelect** is released, the anchor must not move, all the elements between the anchor and the release position must remain in the selection, and the location cursor must be at the location where **BSelect** is released.

The user can extend the range selection using **BExtend**. **BExtend** is usually bound to **MShift BSelect**. When **BExtend** is pressed, the selection must be extended according to one of the following extension models:

Reselect

The selection must extend to include all elements between the anchor and the pointer. When **BExtend** is released, the anchor must not move, all the elements between the anchor and the release position must remain in the selection, and the location cursor must be on the last element selected or at the position of the release for graphics-like collections.

Enlarge Only

The selection can only be enlarged. The selection must extend to include all elements between the old selection and the pointer. In the graphics-like collections, the extended range must be determined by the smallest rectangle that encloses the old selection and the pointer. When **BExtend** is released, the position of the anchor is irrelevant, all the elements selected must remain

in the selection, and the location cursor must be on the last element selected or at the position of the release for graphics-like collections. *

Balance Beam A balance point is defined at the midpoint of the current selection. When **BExtend** is pressed on the opposite side of the balance point from the anchor point, this model must work exactly like the reselect model. When **BExtend** is pressed on the same side of the balance point as the anchor point, this model must move the anchor point to the other side of the selection, and then must work exactly like the reselect model. In graphics-like collections, balancing the anchor point is done separately in each dimension. When **BExtend** is released, all elements must remain in the selection, the anchor must remain in its last position, and the location cursor must be on the last element selected or at the position of the release for graphics-like collections. *

Text-like collections should use the balance beam model. The extension model used by other collections depends on the purpose of the collection.

In the range selection model, clicking **BToggle** can move the location cursor to the position of the pointer without affecting the current selection. **BToggle** is usually bound to **MCtrl BSelect** as shown in the following table.

Table 3-1. Range Selection Bindings

Virtual Button	Default Bindings
BExtend	MShift BSelect
BToggle	MCtrl BSelect

3.1.5 Mouse-Based Discontiguous Selection

The discontiguous selection mechanism allows the user to make multiple simultaneous range selections. It behaves very similar to the range selection model. The description of this selection model assumes that the drag-and-drop model is not in use in the collection. The drag-and-drop model and its effect on this selection model are described in Section 3.2.1.

The behavior of **BSelect** in the discontiguous selection model is exactly the same as in the range selection model. In the discontiguous selection model, when **BSelect** is pressed all elements in the collection must be deselected. If **BSelect** is clicked on a selectable element, it must be selected. Pressing **BSelect** must also set an anchor on the element, or at the position where **BSelect** was pressed. As **BSelect** is dragged through the collection, every element between the pointer and the anchor must be included in the selection. When **BSelect** is released, the anchor must not move, all the elements between the anchor and the release position must remain in the selection, and the location cursor must be on the last element selected. After the anchor point is set with **BSelect**, **BExtend** must work exactly as in the range selection model.

The primary difference between the range selection model and the discontiguous selection model is the use of **BToggle**. In the range selection mode, **BToggle** is only used to move the position of the location cursor without disrupting the current selection. In the discontiguous model, **BToggle** is used to toggle the selection state of an element or a range of elements.

If the current selection is null, **BToggle Click** can leave the anchor alone and leave all elements deselected. Otherwise, if **BToggle** is clicked, the anchor must move to that point, and if **BToggle** is clicked on an element, the selection state of that element must be toggled, and the anchor must be placed on that element. In any case, the location cursor must move to the point at which the click occurs.

In list-like collections, **BToggle Drag** places the anchor on the element at which **BToggle** is pressed and toggles the selection state of that element. In text and graphics-like collections, **BToggle Drag** leaves the anchor at the point at which **BToggle** is pressed, but never on an element. In any case, the location cursor is left at

the point at which **BToggle** is released, and the selection state of the elements in the range dragged out by **BToggle** are toggled based on one of the following toggle models:

Anchor Toggle — For collections where the anchor is on an element rather than between elements, each element in the extended range, regardless of its current selection state, must be set to the selection state of the element with the anchor. That is, if the element with the anchor is selected, all elements in the extended range are also selected; if the element is unselected, all the elements in the extended range are deselected. *

Text Toggle — For collections where the anchor is always between two elements, as in Text, each element in the extended range must be set to the inverse of the selection state of the element within the extended range that is immediately adjacent to the anchor at the time that **BToggle** was pressed. *

Full Toggle — The selection state of all elements in the extended range must be toggled. *

If the anchor is on an element, anchor toggling must be used; otherwise, text or full toggling must be used. *

Once **BToggle** has been used to toggle a selection, the toggled range may be extended by **BExtend**. The extended range can be determined by any of the extension models described in Section 3.1.4; that is, Reselect, Enlarge Only, or Balance Beam. Once the extended range is determined, the selection state of elements added to the range must be determined by the toggle model as described above. If **BExtend** removes elements from the extended range, they should revert to their state prior to the last use of **BToggle**. * *

3.1.6 Keyboard Selection

Selections can be made by using the keyboard as well as the mouse. There are two different keyboard selection models available, normal mode and add mode.

Normal mode is used for making simple contiguous selections from the keyboard. In normal mode, keyboard navigation must never leave the location cursor disjoint from the current selection. In collections where the location cursor is on an element, the element with the location cursor is ordinarily selected; if it is not, pressing **KSelect** moves the selection to the cursored element. *

In normal mode, if the location cursor is disjointed from the selection, editing or inserting at the location cursor should deselect the selection and can move the anchor to the cursor location.

Add mode is used for making discontiguous selections from the keyboard. Even in collections that do not support discontiguous selections, add mode allows the selection to be unaffected by keyboard navigation.

Browse selections must only support normal mode. Single and multiple selections must only support add mode. Range and discontiguous selections must support both modes. The default mode for range and discontiguous selections must be normal mode. **KAddMode** must switch between add mode and normal mode in range and discontiguous selections. When switching from normal to add mode, if the cursored element is the only element selected in the collection, it must be deselected. Mouse-based selection must not change based on the keyboard selection mode. * * * * * * * * * * *

In editable components, add mode is a temporary mode that must be exited when an operation is performed on the selection, or when the selection is deselected. * *

The usual bindings of the virtual keys used in this section are shown in the following table.

Table 3-2. Keyboard Selection Bindings

Virtual Key	Default Bindings
KSelect	**<Select>** **<Space>** **MCtrl <Space>**
KExtend	**MShift KSelect**
KAddMode	**MShift <F8>**

3.1.6.1 Keyboard-Based Single Selection

Collections using the single selection model must use add mode. In add mode, the navigation keys for the collection must move the location cursor independent from the selected element. If **KSelect** is pressed on an unselected element, it must select the element with the location cursor, and must deselect any previous selection in the collection. If **KSelect** is pressed on a selected element, it must deselect the element with the location cursor. *

3.1.6.2 Keyboard-Based Browse Selection

Collections using the browse selection model must use normal mode. The navigation keys move the location cursor and the selection. If the application has deselected all elements or if the cursor is left disjointed from the selection, **KSelect** selects the cursored element and deselects any other elements. *

3.1.6.3 Keyboard-Based Multiple Selection

Collections using the multiple selection model must use add mode. The navigation keys for the collection must move the location cursor independent from the current selection. **KSelect** on an unselected element must add the element to the current selection. **KSelect** on a selected element must remove the element from the current selection.

3.1.6.4 Keyboard-Based Range Selection

Collections using the range selection model must initially use normal mode. This allows a keyboard-based behavior similar to dragging with the mouse.

In normal mode, navigation keys must move the location cursor and deselect the current selection. If the cursor is on an element, it must be selected. The anchor must move with the location cursor. However, in text-like collections, a different model can be used in which the anchor must instead remain at its current location, except that, if the current selection is null, the anchor must be moved to the location of the cursor prior to navigation.

KSelect must move the anchor to the cursor, deselect the current selection, and, if the cursor is on an element, select the element. Unless the anchor is on a deselected item, **KExtend** must extend the selection from the cursor to the anchor, based on the extension model used by **BExtend**.

Using **MShift** in conjunction with the navigation keys must extend the selection. If the current selection is null, or the cursor is disjointed from it, the selection must be deselected, and the anchor must be moved to the cursor. The cursor must then be moved according to the navigation keys, and the selection must be extended based on the extension model used by **BExtend**.

In add mode, **KSelect** and **KExtend** must work exactly as in normal mode. However, ordinary navigation must simply move the location cursor, but must leave the anchor unchanged. Shifted

navigation must move the location cursor according to the * navigation keys, and the selection must be extended based on the * extension model used by **BExtend**.

3.1.6.5 Keyboard-Based Discontiguous Selection

Collections using the discontiguous selection model must initially * use normal mode. In the discontiguous selection model, add mode is not only a separate model of interaction, but an extension to the model that allows the user to select discontiguous elements.

In normal mode, keyboard-based discontiguous selection behaves identically to keyboard-based range selection. In add mode, * **KSelect** must move the anchor to the location cursor and initiate * toggling. If it is on an element, the selection state of that element * must be toggled, but the selection state of all other elements must * remain unchanged. **KExtend** and shifted navigation operations * must extend the selection between the anchor and the location * cursor using one of the toggle mechanisms described in Section * 3.1.5.

3.1.7 Canceling a Selection

KCancel must cancel or undo any incomplete drag operation used * for selection, including selection dragging using the drag-and-drop * model described in Section 3.2.1. Once **KCancel** is pressed to * cancel a drag operation, the application must ignore subsequent key * and button releases until after all buttons and keys are released. **KCancel** is usually bound to **<Escape>** or **<Cancel>**.

3.1.8 Selecting and Deselecting All Elements

There are two special keyboard-based selection mechanisms for selecting (via **KSelectAll**) and deselecting (via **KDeselectAll**) all the elements in a collection. **KSelectAll** in a collection using * multiple, range, or discontiguous selection must select all the * elements in a collection, place the anchor at the beginning of the * collection, and leave the location cursor at its previous position.

Deselection differs between add mode and normal mode. In add * mode, **KDeselectAll** must deselect all the elements in a collection. * In normal mode, **KDeselectAll** must deselect all the elements in a * collection, except the element with the location cursor if the * location cursor is being displayed. In either case, it must leave the * location cursor at its current location and move the anchor to the * location cursor. An application can deselect all elements in a collection regardless of the state of add mode or the selection model.

KSelectAll and **KDeselectAll** are usually bound as shown in the following table.

Table 3-3. KSelectAll and KDeselectAll Bindings

Virtual Key	Default Bindings
KSelectAll	MCtrl </>
KDeselectAll	MCtrl <\>

3.1.9 Using Mnemonics for Elements

Collections can also support mnemonics associated with its * elements if the elements have labels. Typing a mnemonic while the * collection has the keyboard focus must be equivalent to moving the * location cursor to the element and typing **KSelect**.

3.2 Selection Actions

Once a selection is made, the elements of the selection can be acted upon using selection actions. Some common selection actions are delete, cut, copy, and paste, although your application is not limited to these. Selection actions should be available in editable collections. This section describes the following ways of using selection actions.

- The drag-and-drop model
- Using primary selection
- Using quick transfer
- Using keyboard selection actions

3.2.1 The Drag-and-Drop Model

Some collections can support the drag-and-drop model. The drag-and-drop model allows the user to move or copy the selected elements by dragging them to a new location.

Collections can allow their elements to be dragged and dropped within the same collection and different collections. When the dragged elements are dropped outside of the original collection, an appropriate representation of the dragged elements can be moved or copied to the destination collection, if the destination supports dropping of that type of element.

The appropriate representation of dropped elements is determined by the collection in which the elements are dropped. For example, if a graphic element representing a file were dropped in a Text component, it could be appropriate to paste the name or the contents of the file into the Text component. Even collections that do not support dragging of their own elements can support drops of elements from other collections.

If a component implements dragging, it should use **BDrag** to perform the drag operations. Some implementations may choose to use **BSelect** instead of **BDrag** because of limited mouse buttons, consistency with existing interfaces, or user preference. When *

BSelect is used for dragging operations, its use must supersede the use of **BSelect Drag** in the selection models described in Section 3.1. Applications that implement the drag-and-drop model using **BSelect** must also support the use of **BDrag** instead, if it is available. *

When using **BDrag** for dragging elements, if **BDrag** is pressed on a selected element or within a linear range of selected elements, it must drag the entire selection until it is released. If **BDrag** is pressed outside of a selected range in a text-like collection, it must do a quick transfer instead of dragging the element. Quick transfer is described in Section 3.2.3. If **BDrag** is pressed on an unselected element in a list-like collection, it can either do a quick transfer or a dragging operation. The behavior of **BDrag Press** in a graphics-like collection is unspecified. It can either do a quick transfer, a dragging operation, or pan the visible region. *

A drag operation may either copy or move the selected elements. If the elements are copied, a copy or representation of the elements must be pasted into the destination collection, but the original elements must remain unchanged. If the original elements were selected, they must remain selected. *

If the elements are moved, a copy or representation of the elements must be pasted into the destination collection, and the original elements must be deleted. If the original elements were selected, the copy of the elements must become selected if the destination supports selection. *

The type of the elements dragged, and the source and destination of the drag-and-drop operations determine whether the operation causes a move or a copy. The following rules can be user customizable. If not customized, the rules that must be followed are as follows. *

- Dragging from a noneditable component must be treated as a copy. *
- Dragging to a transport device, like a printer or mailbox, must be treated as a copy. *

- Dragging to a removal device, like a wastebasket or shredder, must be treated as a move. *
- Dragging text to or from a Text collection must be treated as a copy. *

In other cases the operation must be treated as a move. Pressing **MCtrl** with the drag operation must make the operation a copy. Pressing **MAlt** with the drag operation must make the operation a move. * * *

When the drag-and-drop model is implemented using **BSelect**, **BSelect Drag** on a selected element or within a linear range of selected elements must initiate a drag operation on the entire selection. **BSelect Drag** on an unselected element must select the element, deselecting all other elements, and initiate a drag operation. * * * * * * *

A component may use **BSelect** for the drag-and-drop model, but not **BToggle**. If **BToggle** is used for the drag-and-drop model, **BToggle Drag** on a selected element must remove the element from the selection. **BToggle Drag** on an unselected element must add the element to the selection and initiate a drag operation on the entire selection. Components that implement the drag-and-drop model using **BToggle**, must also support the use of **BDrag** instead. * * * * * * *

3.2.2 Using Primary Selection

Each collection maintains its own selection. However, there must only be a single primary selection. The primary selection is the selection on which selection actions, like cut and copy, are performed. If more than one selection is highlighted, the primary selection must be the selection that is most visually highlighted. The primary selection is associated with the collection it is in, and it is possible for the primary selection to not contain any elements. In this case, the primary selection is said to be null. * *

There are two models for how the primary selection moves among collections:

- Focus-based, where the primary selection must move to a collection whenever the collection gets the input focus.
- Selection-based, where the primary selection must move to an editable collection whenever a nonnull selection is explicitly started in that collection using either **BSelect**, **BExtend**, **BToggle**, **KSelect**, **KExtend**, **KSelectAll**, or **KReselect**. Clicking **BSelect** or **BToggle**, but not on an element, must not move the primary selection to that component. The primary selection is only moved when a nonnull selection is made by clicking **BSelect** or **BToggle**. When **BExtend** is clicked, an implementation can move the primary selection to the component even if the resulting selection is null. Dragging **BSelect**, **BExtend**, or **BToggle** must move the primary selection to the component even if no elements are ultimately selected.

When a collection loses the primary selection, it should remember what was selected, including the elements contained in it and the position of the anchor. Each collection can provide **KReselect**, which must restore the collection's selection as the primary selection. **KReselect** is usually bound as shown in the following table.

Table 3-4. KReselect Bindings

Virtual Key	Default Bindings
KReselect KSelect	MCtrl MShift KSelect <Select> <Space> MCtrl <Space>

The selection-based, primary selection model should be used whenever possible because when using the focus-based model, it is impossible to perform primary transfer between different components for either a mouse user with a pointer focus policy, or for a keyboard user. Primary transfer is used to cut or copy the primary selection to a new location without using the clipboard.

Clicking **BPrimaryPaste** must paste the primary selection to the position of the pointer and set the location cursor there. The type of

elements in the primary selection, and the source and destination of the primary selection determine whether **BPrimaryPaste** causes a move or a copy in the same manner as the drag-and-drop model described in the previous section.

Clicking **BPrimaryCopy** must copy the primary selection to the * position of the pointer and set the location cursor there. Clicking * **BPrimaryCut** must cut the primary selection, paste it at the * position of the pointer, and set the location cursor there.

Keyboard-based primary transfer, **KPrimaryPaste**, * **KPrimaryCopy**, and **KPrimaryCut** must behave like the mouse- * based primary transfer, except that the primary selection is pasted at * the position of the location cursor instead of the pointer.

If a collection has a fixed insertion point (as in append-only text), then primary transfer can paste the selection at that point.

Primary transfer is usually bound as shown in the following table.

Table 3-5. Primary Transfer Bindings

Virtual Key or Button	Default Bindings
BPrimaryPaste	BDrag
BPrimaryCopy	MCtrl BPrimaryPaste
BPrimaryCut	MAlt BPrimaryPaste
KPrimaryPaste	<vendor defined>
KPrimaryCopy	MCtrl KPrimaryPaste MAlt KCopy
KPrimaryCut	MAlt KPrimaryPaste MAlt KCut
KCut	MShift KDelete <Cut>
KCopy	MShift KInsert <Copy>
KDelete	<Delete>
KInsert	<Insert>

3.2.3 Using Quick Transfer

Quick transfer is used to make a selection, and then immediately cut or copy that selection to the destination cursor. In primary transfer, the user makes a selection, then picks a destination. In quick transfer, the user picks the destination first, then makes a selection to be pasted there. This allows multiple selections to be quickly pasted at the same location. Note that quick transfer does not follow the object-action selection model.

The **destination cursor** is the location where selections will be pasted.

- In text-like collections, the selection must be pasted in the space before the destination cursor. *
- In list-like collections, the selection must be pasted before the element with the destination cursor. *
- In graphics-like collections, the position at which the selection is pasted and the resulting position of the destination cursor are unspecified.

The destination cursor does not need to be in the same collection in which the quick transfer is invoked. The destination cursor must be in the component in which the user last did a selection, insertion, or editing operation. It can either be at the location cursor of that component, or determined independently. * * *

If the destination cursor is determined independently, it must be at the location where selection, insertion, or editing was last done. Furthermore, when the current selection is null, the anchor must be kept at the same location as the destination cursor. The destination cursor should be determined independently in environments in which strong support of keyboard use is expected. * * * *

The position of the destination cursor should be visually marked, typically by a caret, whenever it is not in the collection with the location cursor, or when it is in the same collection, but disjointed.

Quick transfer is generally used as a secondary selection mechanism. That is, it must not disturb the primary selection or affect the clipboard, except when the destination cursor is in or on the boundaries of the primary selection, quick transfer must delete the contents of the primary selection, leaving a null primary * * * *

selection before pasting the transferred elements. Deleting or replacing the contents of the primary selection is called a pending delete.

With the mouse, the quick transfer selection is made by dragging **BQuickPaste**. **BQuickPaste** is usually bound to **BDrag**. Therefore, quick transfer can be unavailable in collections that support the drag-and-drop model. **BQuickPaste Drag** in a collection must select all the elements between the place where **BQuickPaste** was pressed and released, and either cut or copy them to the position of the destination cursor. Whether the operation performs a cut or a copy is determined by the same rules as the drag-and-drop model described in Section 3.2.1. **BQuickCopy Drag** must perform the same action as **BQuickPaste**, but must force a copy operation. **BQuickCut Drag** must perform the same action as **BQuickPaste**, but must force a cut operation.

With the keyboard, **KQuickPaste** must set the quick anchor point to start either a copy or cut operation. Whether the operation performs a cut or a copy is determined by the same rules as the drag-and-drop model described in Section 3.2.1. **KQuickCopy** must set the quick anchor point to start a copy operation. **KQuickCut** must set the quick anchor point to start a cut operation. **KQuickExtend** must perform the quick operation using every element between the quick anchor point and the location cursor.

Quick transfer is usually bound as shown in the following table.

Table 3-6. Quick Transfer Bindings

Virtual Key or Button	Default Bindings
BQuickPaste	**BDrag**
BQuickCopy	**MCtrl BQuickPaste**
BQuickCut	**MAlt BQuickPaste**
KQuickPaste	**<vendor defined>**
KQuickCopy	**MCtrl KQuickPaste**
KQuickCut	**MAlt KQuickPaste**
KQuickExtend	**MShift KQuickPaste**

Quick transfer can be used to transfer static text or graphics that are not ordinarily selectable. For example, a portion of the text label of

a PushButton is not ordinarily selectable, since **BSelect** or **KSelect** activate the PushButton instead. We recommend that developers support quick transfer for all static text and graphics.

While dragging out the quick selection, the range should be highlighted in some way. The highlighting mechanism should be different than that used for the primary selection. In text, the highlight should be underlining. The range determined by the quick anchor and the location cursor should be similarly highlighted during keyboard-based quick selection.

3.2.4 Using Keyboard Clipboard Selection Actions and Deletion

Mouse-based clipboard selection actions are usually performed using the **Edit** Menu of an application. In addition, special bindings are available for keyboard-based clipboard selection. The * keyboard-based clipboard selection actions must be available in * every editable collection. In list-like and graphics-like collections, * **KDelete** and **KBackSpace** must delete the selected elements. In * text-like collections, **KDelete** and **KBackSpace** behave as follows: *

- When the location cursor is in a selection, **KDelete** and * **KBackSpace** must delete the selection. *
- When the selection is null, **KDelete** must delete one character * forward and **KBackSpace** must delete one character backward.
- When the selection is not null and the location cursor is outside of the selection, **KDelete** can either delete the selection or delete one character forward and **KBackSpace** can either delete the selection or delete one character backward.

KCut must cut the selected elements to the clipboard. **KCopy** must * copy the selected elements to the clipboard. **KPaste** must paste the * contents of the clipboard at the position of the location or * destination cursor, except if the collection has a fixed insertion * point (such as append-only text). **KUndo** should undo the last operation on a selection. There can be additional methods of undoing operations on selections. **KCopy** and **KUndo** can also be supported on noneditable collections.

The keyboard-based clipboard selection actions are usually bound as shown in the following table.

Table 3-7. Clipboard Action Bindings

Virtual Key	Default Bindings
KDelete	<Delete>
KCut	MShift <Delete>
KCopy	MCtrl <Insert>
KPaste	MShift <Insert>
KUndo	MAlt <BackSpace>

Pasting in Text must place the pasted text before the cursor. In * graphics, the placement of pasted graphics and the final position of the cursor are specific to the graphics component.

Editable components must also support pending deletion. In * pending deletion, any insert, paste or editing operation first deletes the contents of the current selection. Pending deletion must occur * when the location cursor is in or on the boundaries of the selection.

3.3 Component Activation

Components are used to send information to the underlying application. For example, a Text component is used to send complicated information to the application, but most components, for example, PushButtons are simply used to start an application process. In fact, even a complicated component like Text may need to tell the application that the user is finished entering information.

Components that start some application process are used following the activation model, which is divided into the following areas:

- Basic activation
- Accelerators
- Mnemonics
- Help activation
- Default activation
- Expert activation
- Previewing and autorepeat

3.3.1 Basic Activation

The basic activation model mimics real life button activation in that pressing on a button activates it. The user selects a button with the location cursor, which can be moved among components following the navigation model described in Chapter 2.

- Clicking **BSelect** on the button must activate the button. *
- **KActivate** on the button with the location cursor must activate the button. *
- **KSelect** on a button with the location cursor must activate the button. *

KActivate and **KSelect** are usually bound as shown in the following table.

Table 3-8. Activation Bindings

Virtual Key	Default Bindings
KActivate	**KEnter** **<Return> or <Enter>** **MCtrl KEnter**
KSelect	**<Select>** **<Space>** **MCtrl <Space>**

When **BSelect** is pressed over a button, the appearance of the button must change to indicate that releasing **BSelect** activates the button. If while **BSelect** is pressed the pointer is moved outside of the button, the visual state must be restored. If while **BSelect** is still pressed the pointer is moved back inside of the button, the visual state must again be changed to indicate the pending activation. If **BSelect** is pressed and released within a button, the button must be activated, regardless of whether the pointer has moved out of the button while it was pressed.

An implementation can allow **BSelect Click 2+** (multiple mouse button clicks) to be treated as a single mouse button click on a percomponent basis. In an application where double-clicks are used heavily, this helps prevent the user from unintentionally activating a button twice.

Note that selection and editing components are also activatable, but **KSelect** and **BSelect** are used to select elements within the component. In selection and editing components, only **KActivate** and **BSelect Click 2** can be used to activate the component. As a further restriction, **BSelect Click 2** in a Text component is used to select a word, and in multiline Text components, **KEnter**, the primary binding for **KActivate**, is used to insert a carriage return. **MCtrl KEnter**, the other binding for **KActivate**, must always be available for activating Text components.

3.3.2 Accelerators

An accelerator is a key or key combination that invokes the action of some component regardless of the position of the location cursor when the accelerator is pressed. Accelerators are most commonly used to activate Menu items without first posting the Menu.

If the button with the accelerator is within a primary or secondary window, or within a PullDown Menu system from its MenuBar, then it must be activatable whenever the input focus is in the window. If the button with the accelerator is within a Popup Menu system, then it must be activatable whenever the focus is in the component with the Popup Menu. *

Applications can provide accelerators for any button component. All PushButtons and ToggleButtons must support accelerators in Menus. If a button has an accelerator, the accelerator must be shown following the label of the button. *

3.3.3 Mnemonics

A mnemonic is a single character that can be associated with any component that contains a text label. The label must contain the character, and the character must be underlined within the label. If a label does not naturally contain the character of the mnemonic, the mnemonic must be placed in parentheses following the label. Labels can also be sequentially numbered, and the number can serve as the mnemonic. Labels that are duplicated within an application should be given the same mnemonic. Mnemonics must be case insensitive for activation. Either an uppercase or lowercase letter can be underlined in the label. *

When the location cursor is on a component within a Menu, a MenuBar, or the same field as a component with a mnemonic, typing the mnemonic character must move the location cursor to the component and activate it. If a mnemonic is used for an OptionButton, for a CascadeButton in a MenuBar, or a PushButton that is not in a basic group (that is, not in a Panel, a Menu, or a MenuBar), typing **MAlt** and the mnemonic anywhere in the window *

or its Menus must move the cursor to the component with that mnemonic and activate it. Mnemonics must be supported for PushButtons in a Menu, ToggleButtons in a Menu, and CascadeButtons in a Menu or MenuBar. *

Mnemonics can also be used to select and deselect elements within a component as described in Section 3.1.9.

3.3.4 Help Activation

Help is generally invoked from selections in the **Help** Menu of the MenuBar. In addition, **KHelp** on a component must invoke any context-sensitive help for the component or its nearest ancestor with context-sensitive help available. Within DialogBoxes, applications should provide context-sensitive help for the DialogBox as a whole. **KHelp** is usually bound as shown in the following table. * *

Table 3-9. KHelp Bindings

Virtual Key	Default Bindings
KHelp	<F1> <Help>

3.3.5 Default Activation

A DialogBox should have a default action associated with it. The default action should be the action of a default PushButton in the bottom area of PushButtons of the DialogBox. The default PushButton must be highlighted in some way, usually by displaying a border around it. Default PushButton highlighting can be removed when the focus is outside of the DialogBox. If the DialogBox is nested in a window, the highlighting must be removed when the focus is not the field. The default action of a DialogBox is activated according to the following rules: * *

- When the input focus is on a PushButton other than the default, **KActivate** must activate that PushButton. The highlighting must be removed from the default PushButton, and can be * * *

moved to the new PushButton. When the focus policy is implicit, the changes in highlighting should have a delay to prevent flicker when moving the pointer through components.

- When the focus is on an activatable field control, like List or Text, **BSelect Click 2** or **KActivate** must activate both the field control's corresponding action and the default action of the DialogBox, except where these inputs are used internally by the component (such as **BSelect Click 2** in Text, and **KEnter** in multiline Text.) *

- When the input focus is on any other component, **KActivate** must activate the default action of the DialogBox, but not the action of the component. *

- When the focus is not on any component, **KActivate** must activate the default action of the DialogBox. *

Except in the middle of a drag operation that can be canceled, **KCancel** anywhere in a DialogBox must be equivalent to activating the **Cancel** PushButton in the DialogBox. *

3.3.6 Expert Activation

Some elements, usually PushButtons and ToggleButtons, can have **expert activation** actions associated with them. For example, an icon's expert action is restoring the window. **BSelect Click 2** (that is, double-clicking the element) should activate any expert action for the element. Expert actions should only be available in a Panel of PushButtons or in a Panel of RadioButtons where one of the RadioButtons is always on.

Where expert activation is available, the expert action of all the buttons in the Panel should be the similar. The expert action should include the regular action of the component in a more global manner. For example, a Panel of RadioButtons in a drawing application could include a tool for turning on the erase cursor. Selecting the RadioButton turns on the erase cursor. Double-clicking the RadioButton could erase the drawing area. To support new users and keyboard-only users, expert actions must only be shortcuts to application features available elsewhere. * *

3.3.7 Previewing and Autorepeat

Two special actions can be used with activation: previewing and autorepeat. When **BSelect** is pressed and held over a PushButton or ToggleButton, the application can present information in some way that describes the effect of activating the button. This is called **previewing**. The information must be removed when the user releases **BSelect**. Applications should provide a means to disable previewing for experienced users.

PushButtons may also **autorepeat**. That is, when **BSelect** is pressed and held, the PushButton activates and continues to activate at regular intervals until the PushButton is released. Autorepeating buttons should continue to repeat even when the pointer moves outside the button while the button is pressed; however, applications can suspend the activation of the button until the pointer is moved back inside of the button. While the button is active, it should be drawn in the active state.

Chapter 4

Application Design Principles

This chapter is directed at application designers, rather than the people who write the algorithms of an application, although they are often one in the same. Application designers are the people who design the interface between the application algorithms and the user. In designing user interfaces, an application designer chooses the proper controls or groups of controls, or components, to pass data between the base application and the user, lays out those components naturally, and ensures that user interaction with the components and layout is easy to use.

This chapter describes the three elements of application user interface design, as well as the principles for designing new user interface components. The four major sections of this chapter are as follows:

- Choosing components
- Layout
- Interaction
- Component design

4.1 Choosing Components

The user interface of an application is made up of components that perform two simple functions: presenting the application information to the user, and allowing the user to enter data for the application. Components can be divided into six types: basic controls, field controls, basic groups, layout groups, framing groups, and DialogBox groups. The groups are containers for controls and groups of controls. Groups can be nested.

The component types that should be available in a user interface toolkit are described in the following lists. The details of these components are described in the reference section in Chapter 7 of this guide. The correspondence of the these components to widgets available in the OSF/Motif toolkit is described in Appendix B.

- **Basic Controls**

Separator	Draws a separating line within windows and between Menu items. A Separator does not allow application interaction.
Label	Displays static text and images. A Label presents application information to the user.
PushButton	A button used to activate an operation. A PushButton contains a label that indicates the operation of the button. The label can contain text or an image.
CascadeButton	A button used to display a PullDown Menu. A CascadeButton contains a label that indicates the Menu to be displayed. CascadeButtons can also contain an arrow graphic after the label to distinguish it from PushButtons and to indicate the direction of the cascading Menu.

OptionButton	A button used to display an Option Menu. An Option Menu allows for a one of many selection. An OptionButton contains a label that indicates the current state of the Option Menu, and a bar graphic to distinguish it from a PushButton.
ToggleButton	A button with two states: on and off. A ToggleButton contains a label that indicates the active state. Normally, preceding the label is a graphic indicator of the state of the ToggleButton.
CheckButton	A ToggleButton in a group of ToggleButtons where any number of the ToggleButtons can be on at a time. The graphic indicator for a CheckButton is usually a filled square to indicate the on state or an empty square to indicate the off state. On color systems, the filled color can be distinct from general application colors to visually distinguish the on state.
RadioButton	A ToggleButton in a group of ToggleButtons where only one of the ToggleButtons can be on at a time. The graphic indicator for a RadioButton is usually a filled diamond or circle to indicate the on state or an empty diamond or circle to indicate the off state. On color systems, the filled color can be distinct from general application colors to visually distinguish the on state.

- **Field Controls**

Sash	Used to set the boundary between two components. A Sash is usually a small square on the boundary between two components. The separated components are called Panes, and a group of Panes, Separators, and Sashes is called a PanedWindow.
Scale	Used to set or display a value in a range. A Scale is usually composed of a slider, moving within an element that indicates the size of the range, and a label that indicates the current value. The position of the slider indicates the value relative to the range. The slider is moved directly by using the mouse pointer or by using the arrow keys. A Scale can also have buttons with arrow graphics for moving the slider with the mouse.
ScrollBar	Used to scroll the visible area of a component. A ScrollBar is usually composed of a slider, moving within an element that indicates the full size of the component, and buttons with arrow graphics for moving the slider with the mouse. The slider indicates the relative position and size of the visible area of the component. The slider is moved directly by using the mouse pointer or by using the arrow keys.
List	Used for selecting elements from a list of elements. A List can allow multiple items to be selected or can be constrained to only allow one item to be selected at a time. A List is usually composed of a vertical list of items. A List can also have both horizontal and vertical ScrollBars for scrolling the visible portion of the list of items.

Text	Used for displaying, entering, and modifying text. There are single line and multiple line variants. Multiline Text can have both horizontal and vertical ScrollBars for scrolling the visible portion of the text area.
Canvas	Used for displaying, entering, and modifying graphics. A Canvas can have both horizontal and vertical ScrollBars for scrolling the visible portion of the drawing area.

- **Basic Groups**

Panel	Organizes a collection of components in a horizontal, vertical, or 2-dimensional layout. A Panel is usually composed of just one type of button.
Menu	Organizes a collection of buttons, Labels, and Separators in a horizontal, vertical, or 2-dimensional layout within a separate Menu window. There are three types of Menus: PullDown, Popup, and Option. A Menu is only available while it is pulled down or popped up.
MenuBar	Organizes a collections of CascadeButtons in a horizontal layout at the top of a MainWindow.

- **Layout Groups**

Composition	Organizes a collection of components in an arbitrary layout.
PanedWindow	A linear grouping of components, Separators, and Sashes. Sashes are used to set the boundary between two components. The separated components are called Panes and can contain any components.

- **Framing Groups**

Frame	Draws framing decorations around a component.
ScrolledWindow	Frames a component and adds ScrollBars for scrolling the visible area of the component.
MainWindow	Organizes the contents of a primary window. A MainWindow frames the client area and can optionally include ScrollBars, a MenuBar, a command area, and a message area.

- **DialogBox Groups**

  A DialogBox group can either be nested within another group or organizes the contents of a secondary window. Although a DialogBox may contain any component, a simple DialogBox is composed of a Label and a Panel of PushButtons for supplying a response to the DialogBox. The OSF/Motif toolkit provides a number of ready-designed DialogBoxes for common uses: CommandDialog, FileSelectionDialog, MessageDialog, PromptDialog, and SelectionDialog. There are also a number of different types of MessageDialogs: ErrorDialog, InformationDialog, QuestionDialog, WorkingDialog, and WarningDialog.

4.1.1 Guidelines for Choosing a Main Component Group

At the highest level, components are organized into MainWindows, Menus, and DialogBoxes. Correctly deciding which component group to use for which parts of an application is one of the most important tasks of application designer.

Every application must contain at least one MainWindow. The * MainWindow can contain a MenuBar, ScrollBars, a command area, a message area, and the client area. The client area of the MainWindow contains the framework of an application. The client area should contain all the components needed to perform the primary actions of the applications. For example, in a text editor

application, the MainWindow usually contains a text editing area, and in a graphics editor application, the MainWindow usually contains a graphics editing area and the components for changing drawing functions. Components that are used constantly throughout the application should be contained in the MainWindows client area. Components that are used intermittently or infrequently should be placed in Menus or DialogBoxes.

The advantages of Menus are that they are readily available, quickly accessed and dismissed, and easy to browse through. The most commonly used Menus should be placed in the MenuBar in the MainWindow to increase these advantages. Because Menus are readily available, and quickly accessed and dismissed, they should be used for components that are frequently used. The time delays of bringing up a DialogBox for frequently used components can greatly reduce user productivity. Because Menus are easy to browse through, they should also be used for components that are commonly accessed by most users.

The advantages of Menus also cause some disadvantages. A Menu is a very short-lived component group. It is displayed only while the user makes a selection. Once the user makes a selection, the Menu disappears. So a Menu is not well-suited for making several selections at once.

Note that some systems contain a component group called a Tear-Off Menu. Once a Tear-Off Menu is pulled down, it can be made to remain posted until specifically dismissed. A Tear-Off Menu allows the user to make multiple selections because it can stay posted after a user selection. A Tear-Off Menu behaves more like a DialogBox than a Menu.

Menus are also modal. That is, while a Menu is posted, the user cannot interact with other elements of the application. Because of this, Menus should not be used for components that the user may want to have available while interacting with other elements of the application.

The other disadvantage of Menus is that they can only contain buttons and Labels. Menus should be used for performing simple actions and setting values, but they are not suitable for more complicated functions like text entry.

DialogBoxes are used for two general purposes: to present information to the user, and to take user input. Applications should use DialogBoxes to present transient information to the user, like warnings, cautions, or conformations to actions. DialogBoxes that present information to the user are called Message DialogBoxes. Message DialogBoxes are often placed on the screen by applications without a request from the user. Because Message DialogBoxes can disrupt a user's work, applications should present constantly updated information in the client area of the MainWindow. Also, applications can present minor cautions, incidental notes, and simple help in the message area of the MainWindow.

The advantages of using DialogBoxes to take user input are that they are long-lived, and can contain any components. Because they are long-lived, they can be used to perform more than one action at a time, and can remain available while the user interacts with other parts of the application. Applications should use user-input DialogBoxes like small applications. DialogBoxes should be used to perform tasks ancillary to the application, where the user can take more than one action before dismissing the DialogBox. Applications should also use DialogBoxes where the task requires more complicated interaction than is available in a Menu, like file selection.

The following table summarizes the suggested uses for MainWindows, Menus, and DialogBoxes.

Table 4-1. Suggested Window Types

For These Cases	Use This Main Component Group		
	MainWindow	Menu	DialogBox
Primary application actions	X	—	—
Ancillary application actions	—	—	X
Components used intermittently	—	X	—
Components used frequently	X	X	—
Components used seldomly	—	—	X
Components accessed by most users	—	X	—
Simple actions	X	X	—
Complex actions	X	—	X
Presenting transient information	—	—	X
Presenting updating information	X	—	—
Presenting minor information	X	—	—

4.1.2 Guidelines for Choosing Interactive Methods

Choosing the correct components for a task is usually simple. In most cases, a task seems to naturally belong to a component; for instance, scrolling a region with a ScrollBar, or choosing a value from a range with a Scale. Each component's general purpose is described in Section 4.1, as well as in the reference section in Chapter 7. In some cases though, it is difficult to decide between two controls or groups of controls that could perform the same task. The following sections give some guidelines for choosing among some common similar components.

- Choosing a single-choice component
- Choosing a multiple-choice component
- Choosing between a Menu and PushButton
- Choosing between Text or Canvas, and a Label

4.1.2.1 Choosing a Single-Choice Component

A single-choice component allows the user to select a single item from a group of items. Applications should use a single-choice component to limit the user to one choice; for example, choosing among available pen widths in a drawing program. The available single-choice components are as follows:

- RadioBox
- OptionButton
- List

You should choose a single-choice component based on the number of items to choose among, the space available for the selection area, and the permanence of the selection. For a small number of items, the best component is a RadioBox or an OptionButton. RadioBoxes should usually contain no more than 5 or 6 items. OptionMenus, which are connected to OptionButtons, should usually contain no more 10 to 12 items. If there is very little space available in your application, you should use an OptionButton over a RadioBox, since the OptionButton takes up very little space.

When the number of items to choose from gets larger than a RadioBox or OptionButton can easily handle, you should use a List. You should also use a List when the items to choose from can change. For application consistency, the choices in a RadioBox or OptionButton should not change, but if they do, you should use an OptionButton over a RadioBox. If a RadioBox or OptionButton choice becomes unavailable, it should be disabled rather than removed. Section 4.3.1.6 describes disabling components in detail.

4.1.2.2 Choosing a Multiple-Choice Component

A multiple-choice control or group of controls allows the user to simultaneously choose multiple items from a group of items. Applications should use multiple-choice components, rather than a

series of single-choice actions, where the user may want to select more than one item from a group. The available multiple-choice components are as follows:

- CheckBox
- List

As with single-choice components, the main factor when choosing which to use is the number of items in the group. For small groups of fewer than seven static elements, applications should use a CheckBox. Applications should otherwise use a List. Note that a List component can be used as a single- or multiple-choice component.

4.1.2.3 Choosing Between a Popup Menu and a Panel or PullDown Menu

Popup Menus should only be considered as shortcuts to application features because they are hidden in the application. Features hidden in Popup Menus are difficult to find for beginning users of an application because they provide no cue to their existence. Applications should use Popup Menus to allow users to remain focused on their work areas, and when there is not enough space in the client area for a Panel of buttons.

Popup Menus contain a set of buttons that can be used in the same way as a Panel of buttons or a PullDown Menu. The primary difference is availability. A Popup Menu is only visible when requested by the user. At all other times it is hidden, providing no cue to its existence. When the user requests a Popup Menu, the application displays it at the location of the pointer. Buttons and PullDown Menus on the other hand are always visible, but users need to move the focus away from their main work to activate them. Another difference is that buttons take up screen space while Popup Menus do not. PullDown Menus provide a good compromise in space and availability versus Popup Menus and Panels.

Applications should use a Panel of buttons when the user makes frequent or multiple selections, and when space in the client area allows. Whenever the choice between Popup Menus and Panels is difficult, applications should favor Panels.

4.1.2.4 Choosing Between Text or Canvas, and a Label

A Label is a simple mechanism for displaying text or graphics. A Label does not have any mechanisms for the user to edit its contents, or the overhead required for editing. Applications should use a Label for displaying text or graphics that do not need user editing. Applications should use a Text or Canvas when the text or graphics needs to be edited. On some systems, the contents of a Label are not selectable. If the user needs to select the contents of some noneditable text or graphics, applications should use a Text or Canvas and disable editing rather than using a Label.

4.2 Layout

The previous section described the components that are available for building an application. This section describes how to combine those components into a coherent application that encourages a user's sense of control. The guidelines presented in this section will help you create applications that are both consistent with themselves and with other OSF/Motif compliant applications.

This chapter discusses the following client area design topics:

- Client areas
- Grouping components
- Menu design
- DialogBox design

Design the layout of your application windows according to the natural use order and the natural scanning order of the people who will be using your application.

First design for the natural use order. Consider the tasks that the user will perform with your application. The components should be positioned so that moving among the components is simple and quick while performing the most common tasks—the less pointer movement, the better. This is also true for keyboard traversal—the fewer keystrokes required to perform a task, the better.

The natural scanning order is most important when arranging small groups of components to help the user find the correct component for the task. You should put the most important and most used commands first. In most cases, this order is from left to right and from top to bottom. For users in right-to-left direction language environments, the natural scanning direction is from right to left and from top to bottom.

4.2.1 Common Client Areas

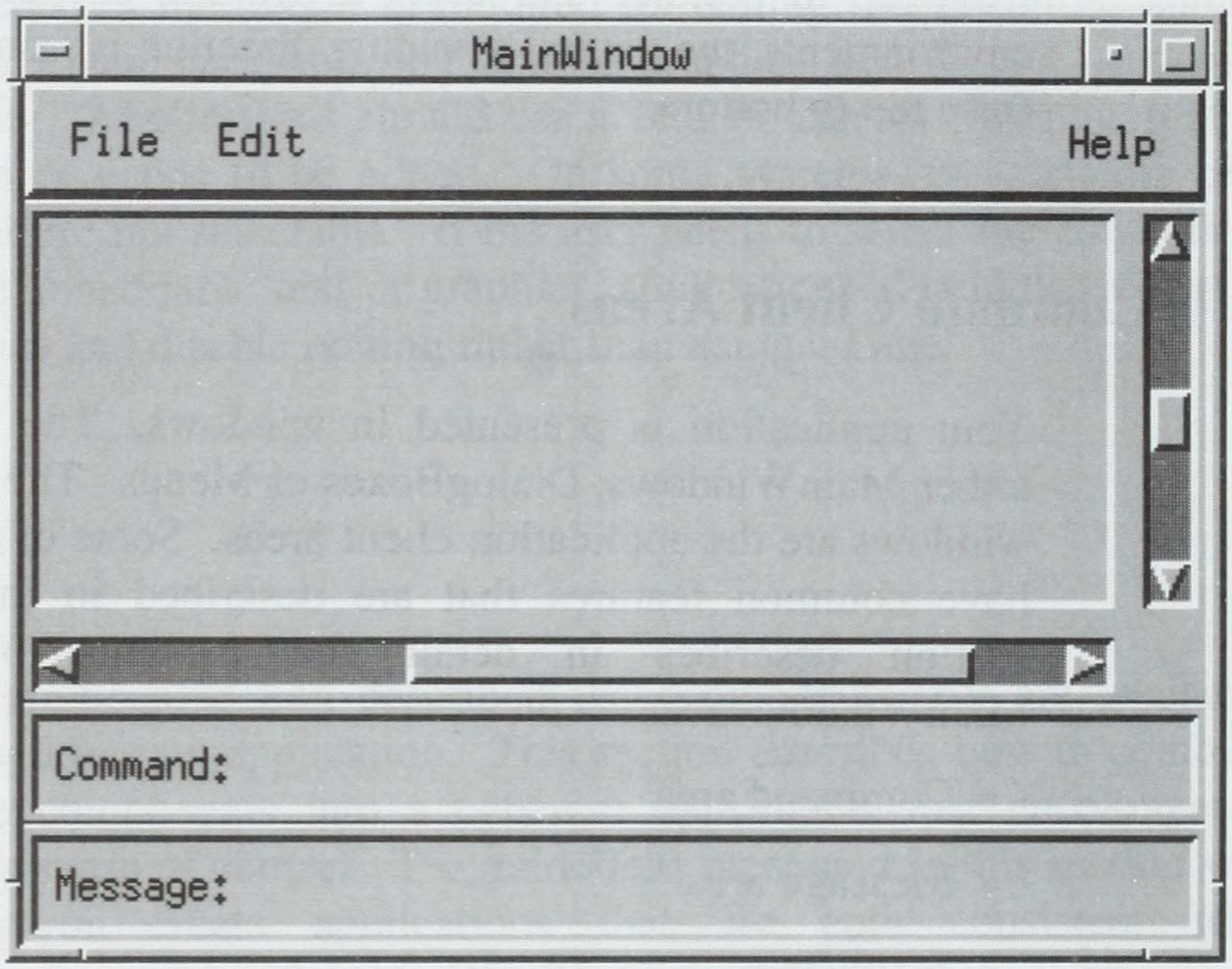
Your application is presented in windows. The windows can be either MainWindows, DialogBoxes or Menus. The contents of these windows are the application client areas. Some of these client areas have common features that are described in this section. This section describes in detail the contents of the common MainWindow areas.

- Command area
- Message area
- ScrollBars
- MenuBar

Following this is a description of some common DialogBoxes. These common client areas provide a familiar base for users new to your application to begin working.

Figure 4-1 shows a MainWindow with its client areas.

Figure 4-1. A Typical MainWindow and Its Common Client Areas



MainWindows contain the framework for your application. In a text editing application, a MainWindow usually contains the text. In a spreadsheet application, a MainWindow usually contains the spreadsheet. In a mail browsing application, a MainWindow usually contains a list of mail messages and an area for viewing a message. Every application must have at least one MainWindow. *

Some applications may want more than one MainWindow; for example, applications with more than one primary function, like the query and update features of a database, and applications with more than one instance of the same primary function, like a graphics editor working on multiple figures. When an application has multiple MainWindows that serve the same primary function, each window should be treated as a separate application. Each window * must close and iconify separately. When an application has * multiple MainWindows that serve different primary functions, each * window must be iconifiable separately, but it is up the application to * decide whether each closes separately, or whether closing one *

window closes the entire application. You should take special care to identify, using the window manager title bar or a prominent internal label, which DialogBoxes belong to each MainWindow.

4.2.1.1 Command Area

Even though user interface applications are oriented toward direct manipulation of graphical elements, your application can provide an area for a user to enter a typed command. This can be particularly useful when updating an existing command-based application to a graphical user-interface-based application.

If your application includes a command area, it should run from border to border across the bottom of the MainWindow, just above the message area, or just below the MenuBar. We recommend that a command area run just above the message area.

4.2.1.2 Message Area

Your application can provide a message area in the MainWindow for presenting application messages. Your application should not use a message area for warnings or messages requiring immediate action. These should be displayed in a DialogBox.

If your application includes a message area, it should run from border to border across the bottom of the MainWindow, below any ScrollBars or command area. Your application can display and remove the message area as needed.

4.2.1.3 ScrollBars

An application can use ScrollBars in its MainWindow to provide a means for viewing an area larger than the MainWindow. Your application defaults should place ScrollBars on the bottom and right sides of the application. (ScrollBar placement can be a user preference.) You should place ScrollBars below the MenuBar and

above any command or message area. ScrollBars should not scroll the MenuBar, command area, or message area. The ScrollBars can be displayed and removed as needed.

4.2.1.4 MenuBar

The MenuBar is a basic group that organizes the most common features of an application. The MenuBar must be a horizontal bar at * the top edge of the application just below the title area of the * window frame. The MenuBar contains a list of Menu topics in CascadeButtons with PullDown Menus connected to them. * Commands must not be included as topics in the MenuBar because * they inhibit Menu browsing. Each Menu topic should have a single-letter mnemonic indicated by underlining.

While we recommended that you include the common Menus in the MenuBar of your application, your choice of Menu titles and items depends on the nature of your application. If your application requires it, you should design more relevant titles and selections, but do not change the meanings of words used in the common Menus.

The following PullDown Menus provide general functions common to most applications. Remember that these Menu guidelines, like all the guidelines in this *Style Guide*, only apply in a left-to-right language direction environment in an English locale. You need to make the appropriate changes for other locales.

File The **File** Menu should contain components for performing actions on the files, such as opening, saving, closing, and printing. It should also contain components for performing actions on the application as a whole, such as quitting. If your application * includes a **File** Menu, it must be the first entry in the * MenuBar, and must be placed to the far left. The **File** Menu should have a mnemonic of **F**.

Edit The **Edit** Menu should contain components for performing actions on the current data of the application, such as an undo action or making global substitutions in a block of text. It should also include

components for interacting with the system clipboard, such as cut, copy, and paste. If your application includes an **Edit** Menu, it must be placed to the right of the **File** Menu. The **Edit** Menu should have a mnemonic of **E**. *

View — The **View** Menu should contain components for changing the user's view on the data. Components in the **View** Menu should not actually change the data. The exact contents of the **View** Menu are application specific, but it may include components that change the appearance of the data, the amount of data that is displayed, or the order in which the data is displayed. If your application includes a **View** Menu, it must be placed to the right of the **Edit** Menu. The **View** Menu should have a mnemonic of **V**. * *

Options — The **Options** Menu should contain components for a user to customize the application. The exact contents of the **Options** Menu are application specific. If your application includes an **Options** Menu, it must be placed to the right of the **View** Menu. The **Options** Menu should have a mnemonic of **O**. * *

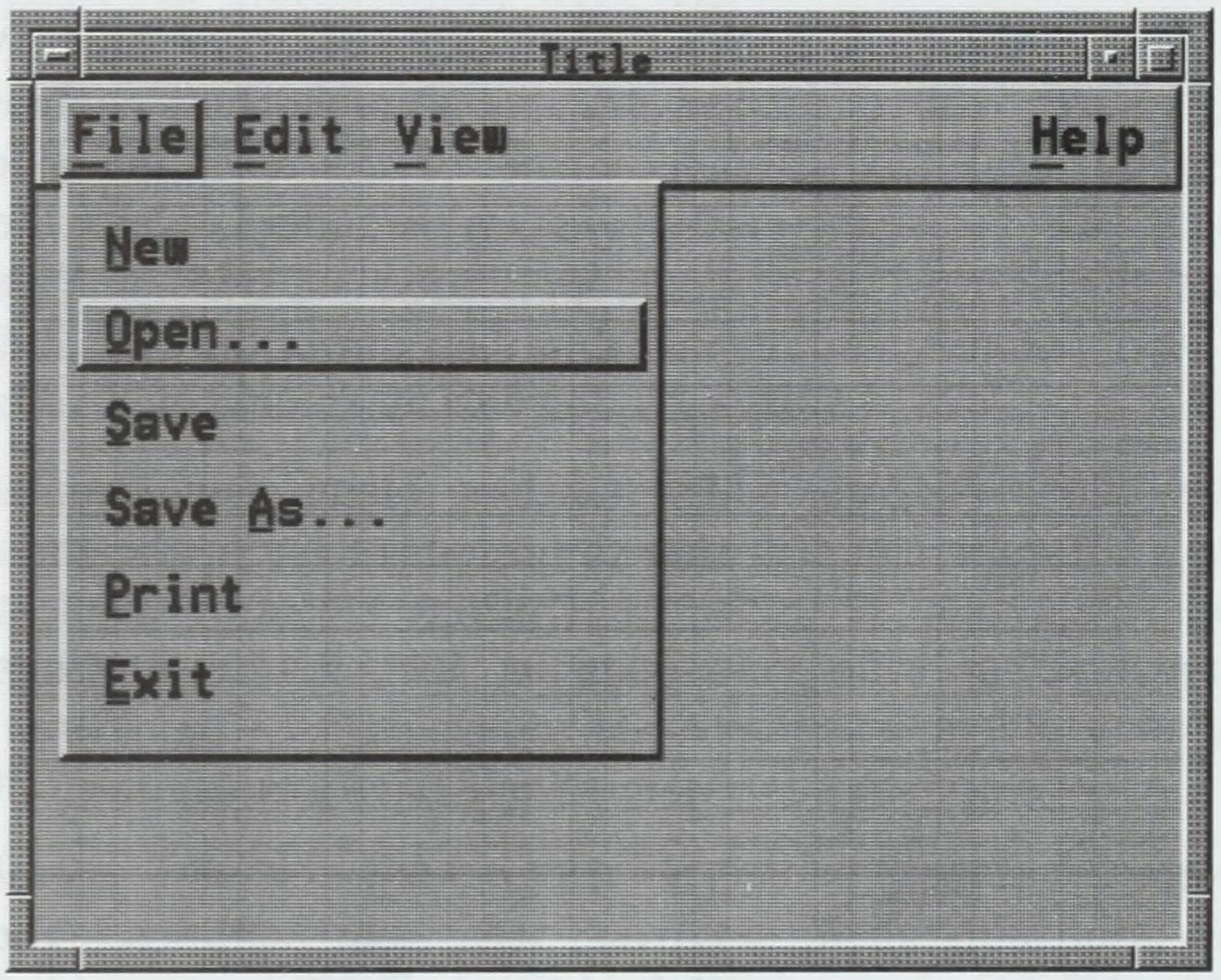
Help — The **Help** Menu should contain components that provide user help facilities. The components in the **Help** Menu usually bring up a DialogBox with help information. Every application should have a **Help** Menu. If your application includes a **Help** Menu, it must be the last Menu item and placed to the far right on the MenuBar. The **Help** Menu should have a mnemonic of **H**. * *

The recommended contents of the **File**, **Edit**, and **Help** Menus follow. The contents of the **View** and **Options** Menus are application specific and are not specified here.

4.2.1.4.1 File Menu Contents

Figure 4-2 shows a sample **File** Menu. The common Menu contents are described following the figure. Note that you should only include those functions actually supported by your application.

Figure 4-2. The File Menu and Its Selections



New Must create a new file. The **New** operation must clear existing data from the client area. If completion of the operation will obliterate current changes to the file, you must display a DialogBox, asking the user about saving changes. This action must have the mnemonic **N**.

Open . . . Must open an existing file. The **Open** operation must prompt the user for the name of the file with a DialogBox. The title bar should be updated with the name of the newly opened file. If completion of the operation will obliterate current changes to the file, you must display a DialogBox, asking the user about saving changes. This DialogBox should be displayed after the user selects a new file. This action must have the mnemonic **O**. *

Save Must save the currently opened file without removing the existing contents of the client area. If the currently opened file has no name, **Save** must prompt for a filename with a DialogBox. This action must have the mnemonic **S**. *

Save As . . . Must save the currently opened file under a new name without removing the existing contents of the client area. The **Save As** operation must prompt for the name of the file with a DialogBox. If the user tries to save the new file under an existing name, **Save As** must alert the user with a DialogBox of a possible loss of data. This action must have the mnemonic **A**. *

Print Must schedule a file for printing. If your application requires specific printing information before printing, the operation must first request that information with a DialogBox, and the entry title must be followed by an ellipses. Printing information can also be specified for the application in the **Options** Menu. This action must have the mnemonic **P**. *

Close Can be supplied in applications that have multiple independent primary windows. This action must not be supplied in applications with a single primary window, or multiple dependent primary windows. This action must only close the current primary window and its associated secondary windows; that is, the window family. This action must have the mnemonic **C**. You can include this action even though it is similar to the **Close** action *

in the window Menu. This ensures that users have a way to close the primary window even if they are not running a compliant window manager. Applications should prompt the user to save any unsaved changes if the action would cause loss of data.

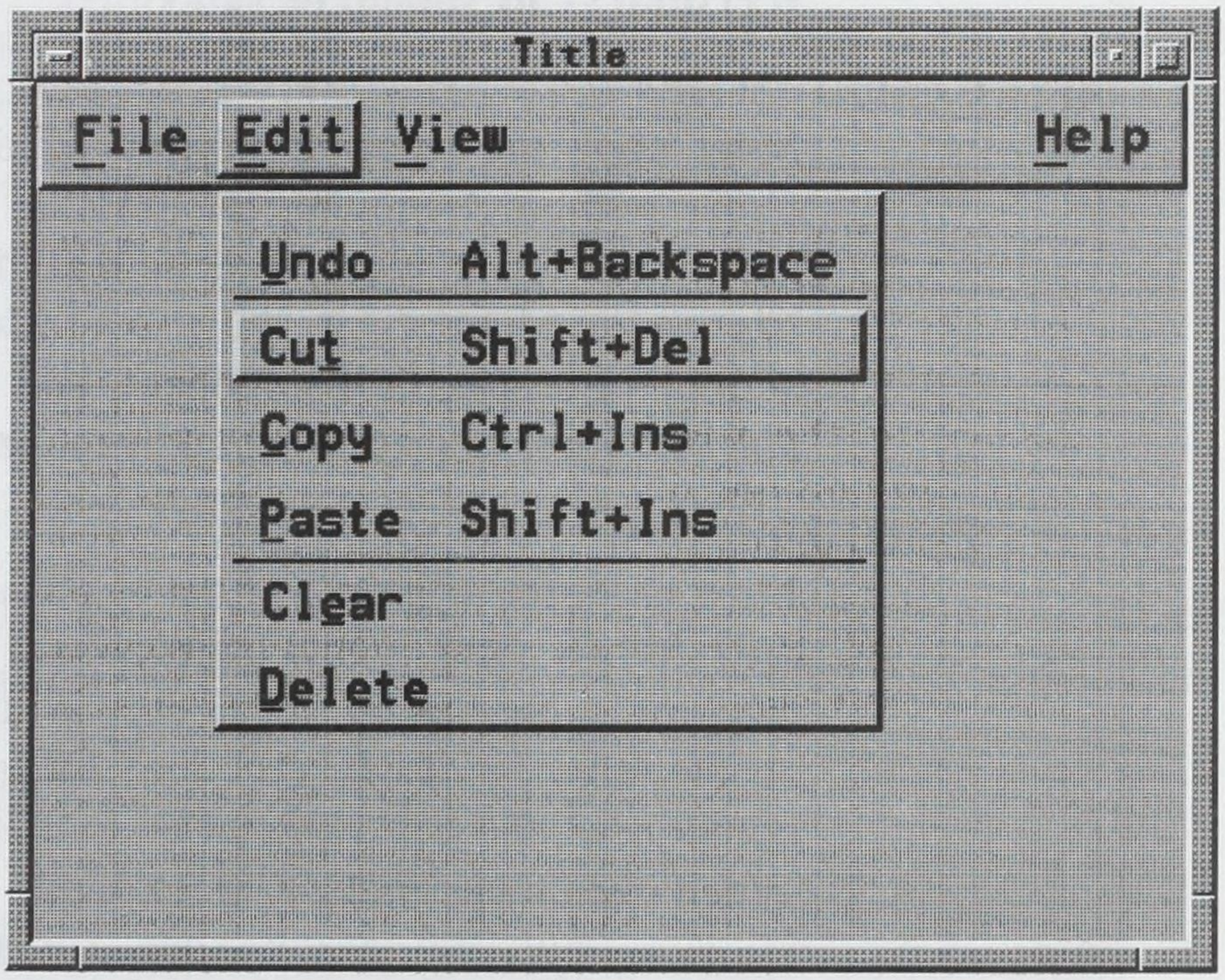
Exit Must end the current application and all windows associated with it. This action is equivalent to closing all primary windows of the application. If completion of the operation will obliterate current changes to the file, you must display a DialogBox, asking the user about saving changes. This action must have the mnemonic **E**. You should include this action even though it is similar to the **Close** action in the window Menu. This ensures that users have a way to end the application even if they are not running a compliant window manager. Applications should prompt the user to save any unsaved changes if the action would cause loss of data. If your application does not have a **File** Menu, put **Exit** at the end of the first PullDown Menu. *

You should include Menu items in the order described. If you add new Menu items, you should insert them near similar elements.

4.2.1.4.2 Edit Menu Contents

Figure 4-3 shows a sample **Edit** Menu. The common Menu contents are described following the figure. Note that you should only include those functions actually supported by your application.

Figure 4-3. The Edit Menu and Its Selections



Undo Alt+Backspace

Must reverse the most recently executed action. To provide a visual cue to the user, the **Undo** selection title should be dynamically modified to indicate what is being undone. For example, if the most recently executed action were a paste, the action name would be **Undo paste**. Your application should be able to undo all of the actions in the **Edit** Menu. This action must have the mnemonic **U**. If this action has an accelerator, it must override the standard undo action for a specific component, and must be **MAlt KBackSpace**. *

Cut Shift+Del

Must remove the selected portion of data from the client area to the clipboard buffer. Your application can determine whether the area that was occupied by the *

removed data is left blank or whether the remaining data is compressed to fill in the space. Usually, graphics applications leave the space blank while text applications compress the remaining text to fill in the space. This action must have the mnemonic **T**. If this action has an accelerator, it must override the standard cut action for a specific component, and both **MShift KDelete** and **KPrimaryCut** should be supported. *

Copy Ctrl+Ins

Must copy the selected portion of data to the clipboard buffer without removing the original data from the client area. This action must have the mnemonic **C**. If this action has an accelerator, it must override the standard copy action for a specific component, and both **MCtrl KInsert** and **KPrimaryCopy** should be supported. *

Paste Shift+Ins

Must paste the contents of the clipboard buffer into a client area. When there is one main editing area, the application can force the paste to occur in that area. When an accelerator is used and the focus is in an editable component, the paste should go to that editable component. When the focus policy is explicit, and when there are multiple possible destinations for the paste, the paste should go to the last editable component with the input focus. When the focus policy is implicit, and when there are multiple possible destinations for the paste, the paste should go to the component with the destination cursor, as described in Section 3.2.3. *

Your application can determine whether the pasted data is reformatted to fit in the client area and whether existing data moves to create room for the pasted data. Text applications usually reformat the pasted text to fit into the margins of the text field, and they will move the existing text to make room for the new text. Graphics applications usually do neither. This action must have the mnemonic **P**. If this action has an accelerator, it must override the standard paste action for a specific component, and both **MShift KInsert** and **KPrimaryPaste** should be supported. * *

Clear Must remove a selected portion of data from the client area without copying it to a clipboard buffer. The remaining data is not compressed to fill the space that was occupied by the cleared data. This action must have the mnemonic **E**. *

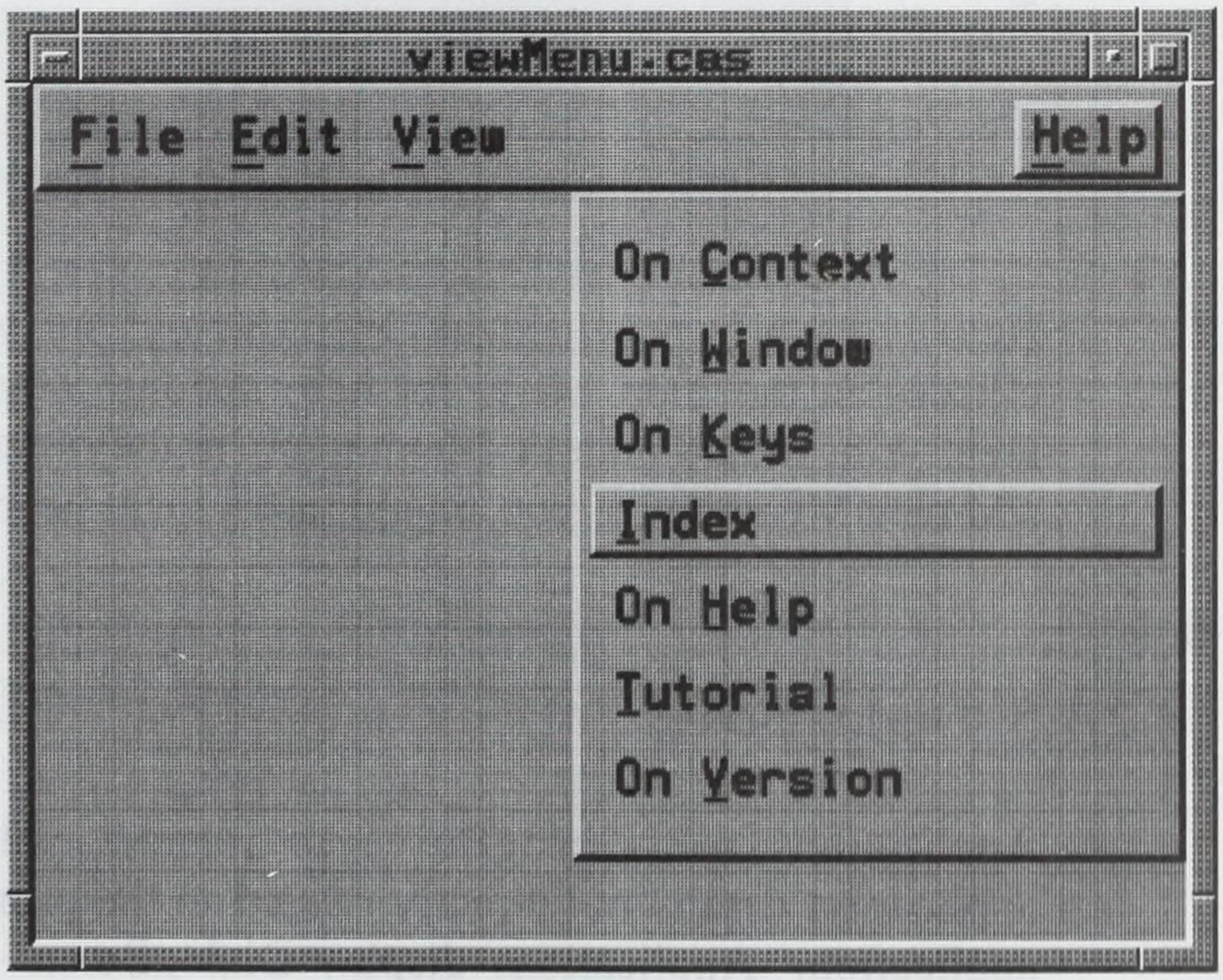
Delete Must remove a selected portion of data from the client area without copying it to a clipboard buffer. The remaining data is compressed to fill the space that was occupied by the deleted data. This action must have the mnemonic **D**. *

You should include Menu items in the order described. If you add new Menu items, you should insert them near similar elements.

4.2.1.4.3 Help Menu Contents

Figure 4-4 shows a sample **Help** Menu. The common Menu contents are described following the figure. Note that you should only include those functions actually supported by your application.

Figure 4-4. The Help Menu and Its Selections



On Context	Must initiate context-sensitive help by changing the shape of the pointer to the question pointer described in Section 2.2.2. When the user moves the pointer to the component help is wanted on and presses **BSelect**, any available context-sensitive help for the component must be presented, and the pointer reverts from the question pointer. This action must have the mnemonic **C**.	* * * * * * * * *
On Help . . .	Must provide information on how to use the application's help facility.	*
On Window . . .	Must provide general information about the window from which help was requested.	*
On Keys . . .	Must provide information about the application's use of function keys, mnemonics, and keyboard accelerators.	* *
Index . . .	Must provide an index for all help information in the application. The index can provide search capabilities.	*
Tutorial . . .	Must provide access to the application's tutorial.	*
On Version . . .	Must provide the name and version of the application. It can provide other information as well.	*

Help is usually provided in DialogBoxes, but you should not use ellipses in the **help** Menu for help actions provided in a message area. You should include Menu items in the order described. If you add new Menu items, you should insert them near similar elements.

4.2.1.5 DialogBoxes

Applications use DialogBoxes to interact with the user about application details not directly related to the primary purpose of the application. Applications display DialogBoxes only when needed to convey a message to the user, or when the user requests it to provide input to the application. They follow the same general

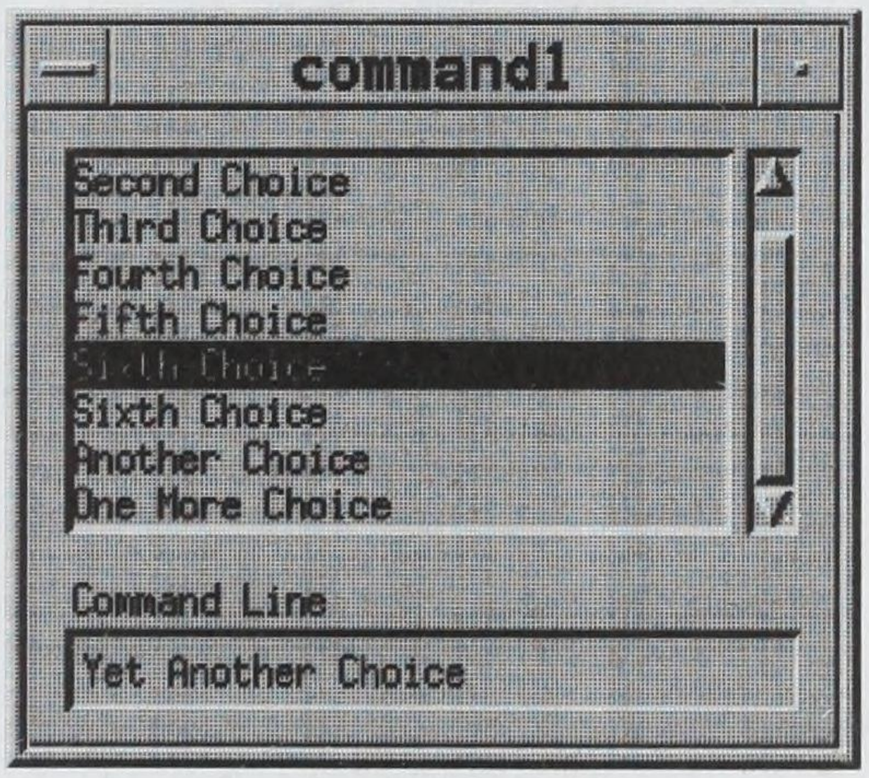
layout guidelines as a MainWindow. Additional guidelines for DialogBox design are given in Section 4.2.4, as well as in the reference section in Chapter 7.

The following DialogBoxes provide general functions common to many applications.

4.2.1.5.1 CommandDialog

A **CommandDialog** should be used to enter keyboard commands. It should not interrupt the user's interaction with the application; that is, it should not be modal. It should include a CommandBox as shown in Figure 4-5.

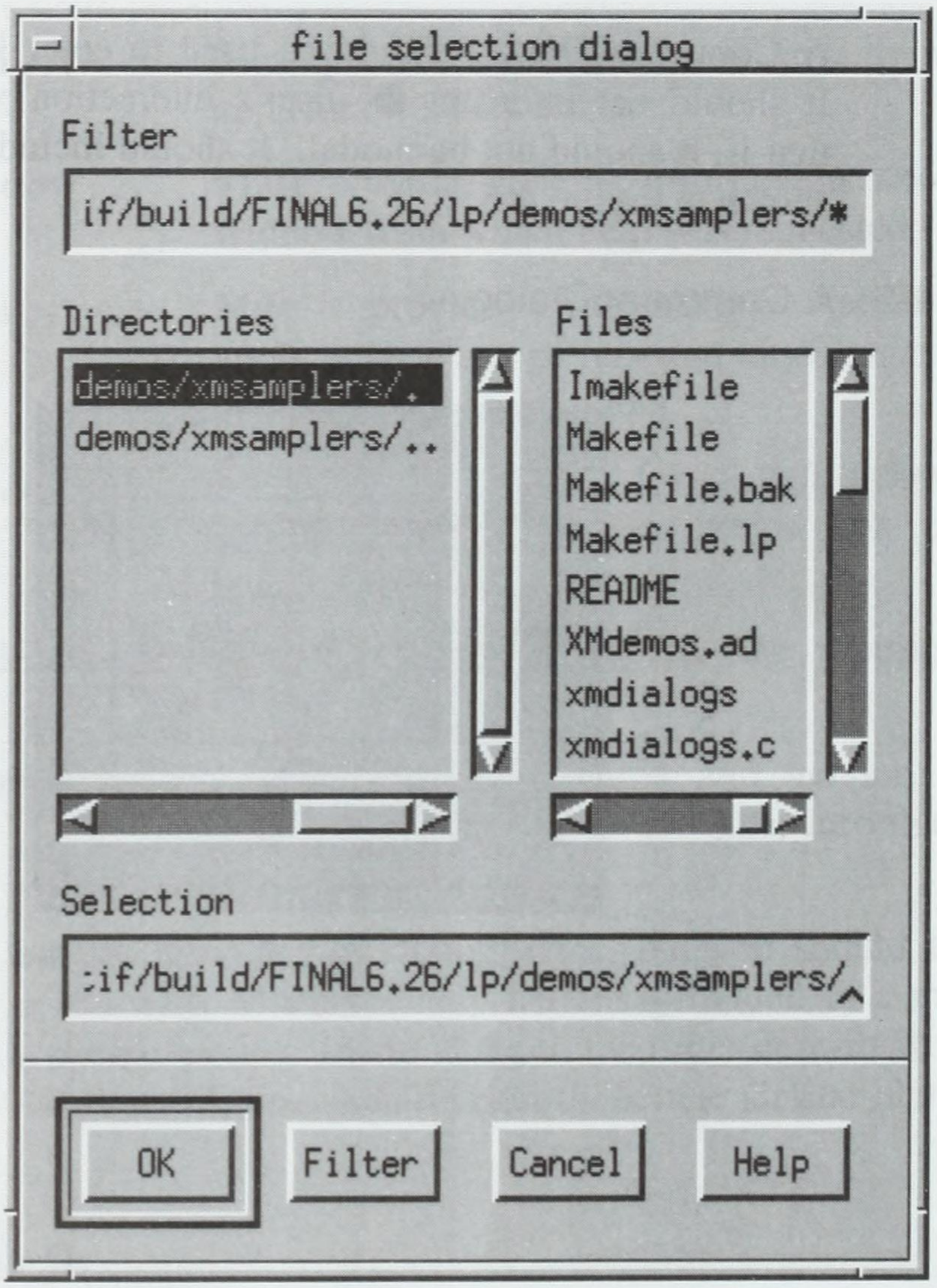
Figure 4-5. A CommandDialog



4.2.1.5.2 FileSelectionDialog

A **FileSelectionDialog** should be used to enter the name of file for processing. It should not interrupt the user's interaction with the application; that is, it should not be modal. It should include a FileSelectionBox as shown in Figure 4-6.

Figure 4-6. A FileSelectionDialog

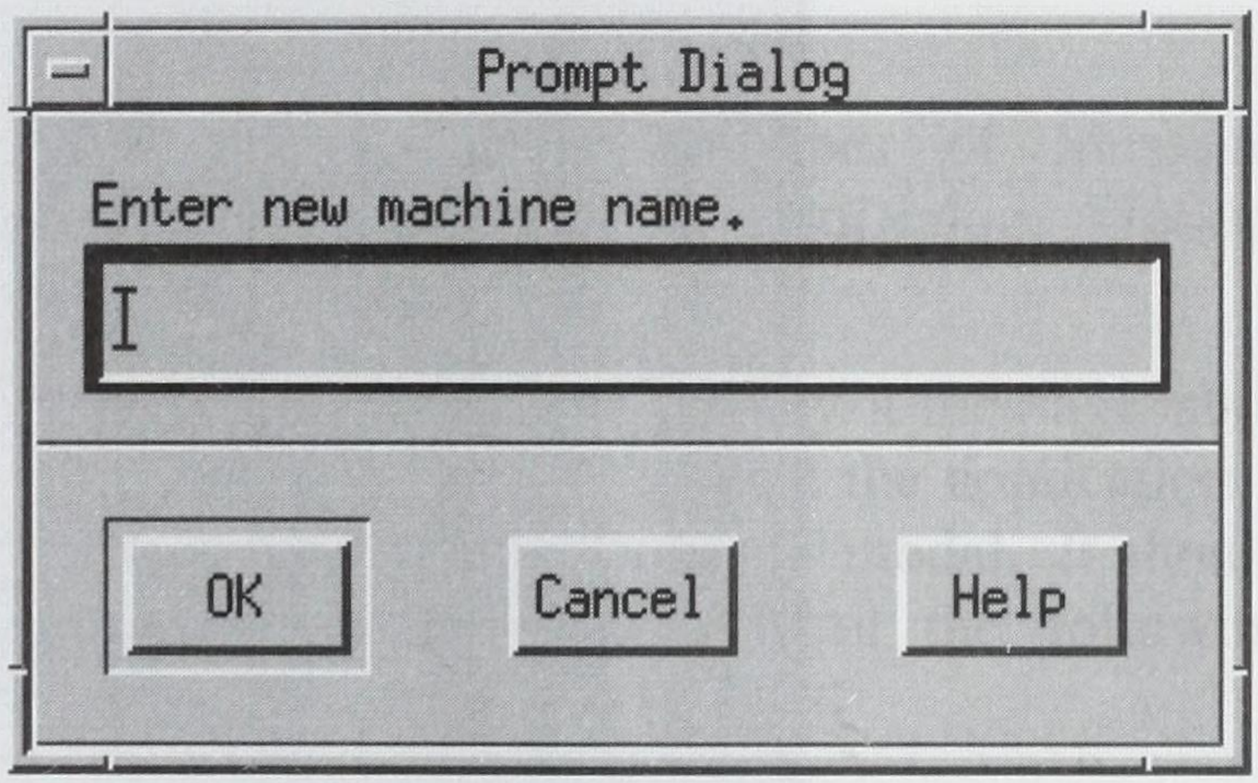


4.2.1.5.3 PromptDialog

A **PromptDialog** should be used to prompt the user for input. It can interrupt the user's interaction with the application; that is, it can be application modal. It should include a message, a text input area, and one of the following button arrangements as shown in Figure 4-7.

OK Cancel
OK Cancel Help
OK Apply Cancel
OK Apply Cancel Help
OK Apply Reset Cancel
OK Apply Reset Cancel Help

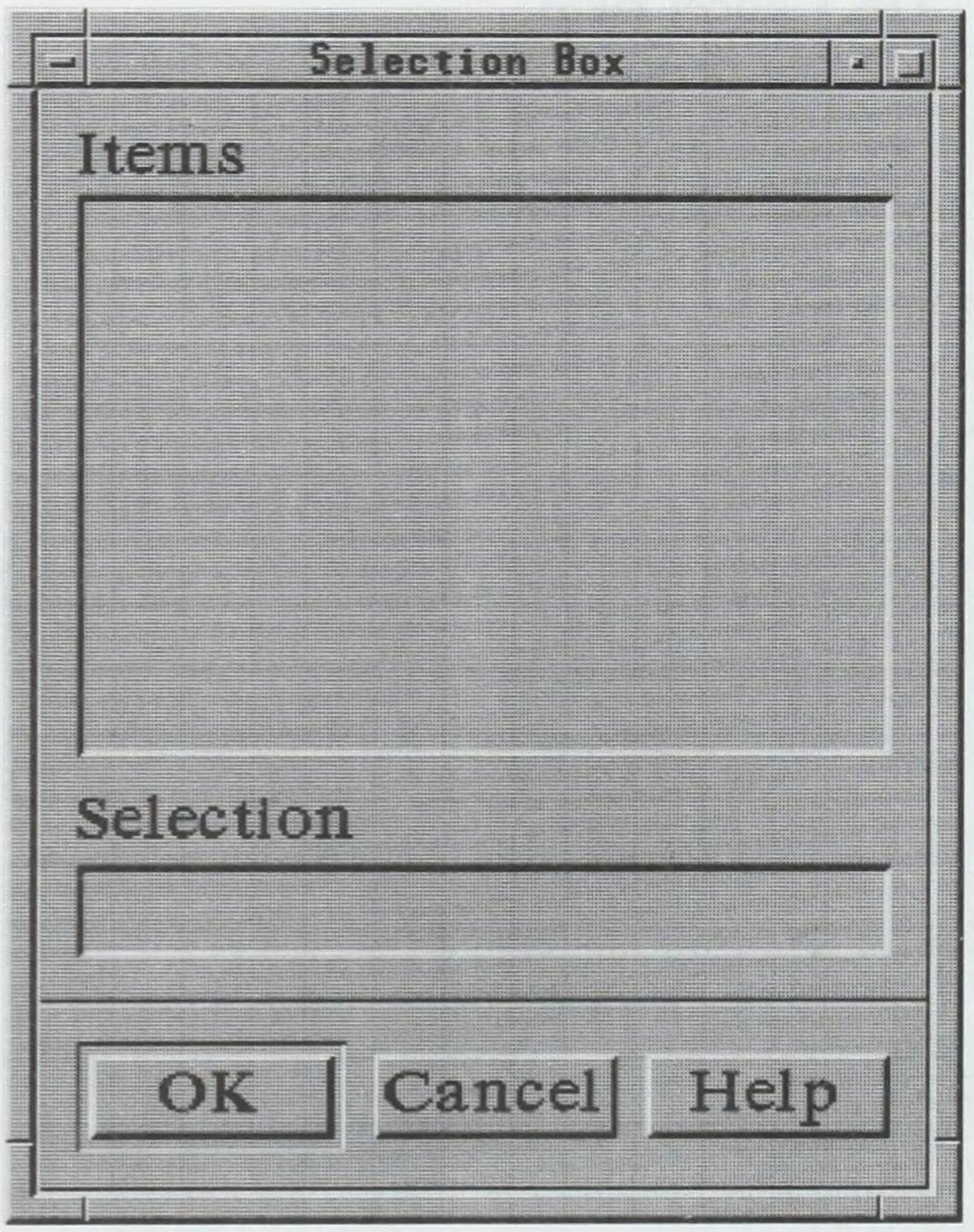
Figure 4-7. A PromptDialog



4.2.1.5.4 SelectionDialog

A **SelectionDialog** should be used to allow a user to make a selection from a list of choices. It can interrupt the user's interaction with the application; that is, it can be application modal. It should contain a SelectionBox as shown in Figure 4-8.

Figure 4-8. A SelectionDialog



4.2.1.5.5 MessageDialog

A **MessageDialog** should be used to convey a message to the user. * It must not interrupt the user's interaction with the application; that * is, it must not be modal. It should include a message, and one of the following button arrangements.

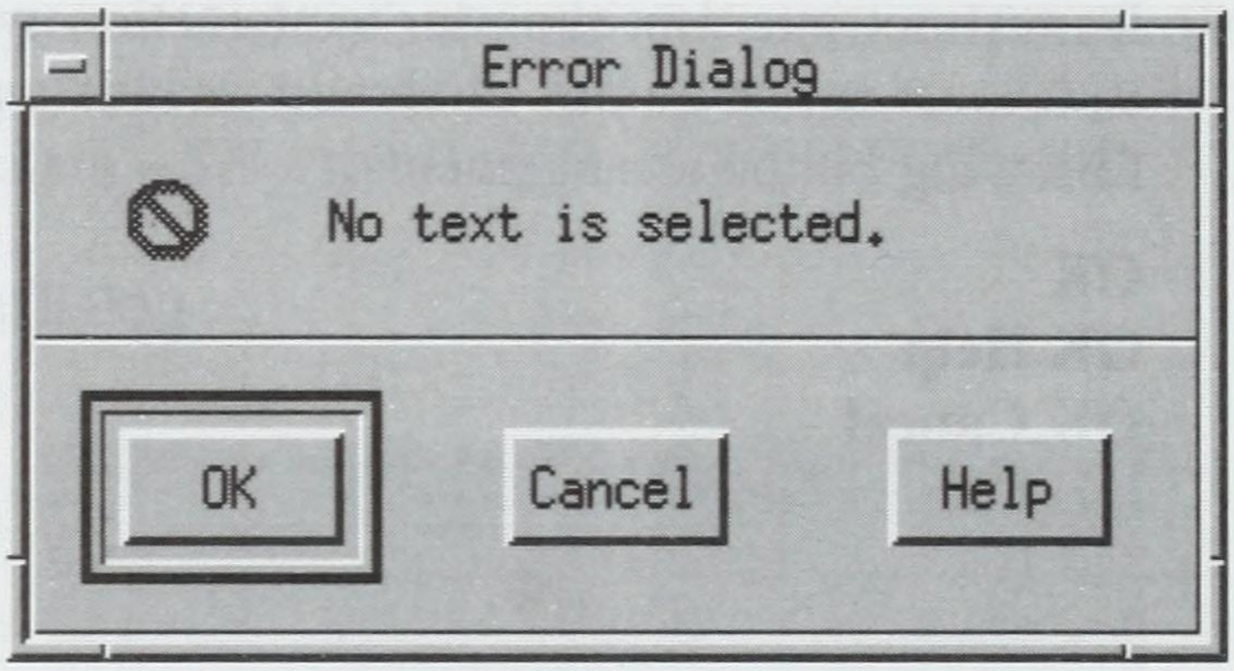
OK
OK Help
OK Cancel
OK Cancel Help
Yes No
Yes No Help
Yes No Cancel
Yes No Cancel Help
Cancel
Cancel Help
Retry Cancel
Retry Cancel Help

There are a number of different types of MessageDialogs: ErrorDialog, InformationDialog, QuestionDialog, WorkingDialog, and WarningDialog.

An ErrorDialog should be used to convey a message about a user error. It should stop user interaction with the application until it is dismissed, that is, it should be application modal. It should include an error symbol, a message, and one of the following button arrangements as shown in Figure 4-9.

OK Cancel
OK Cancel Help

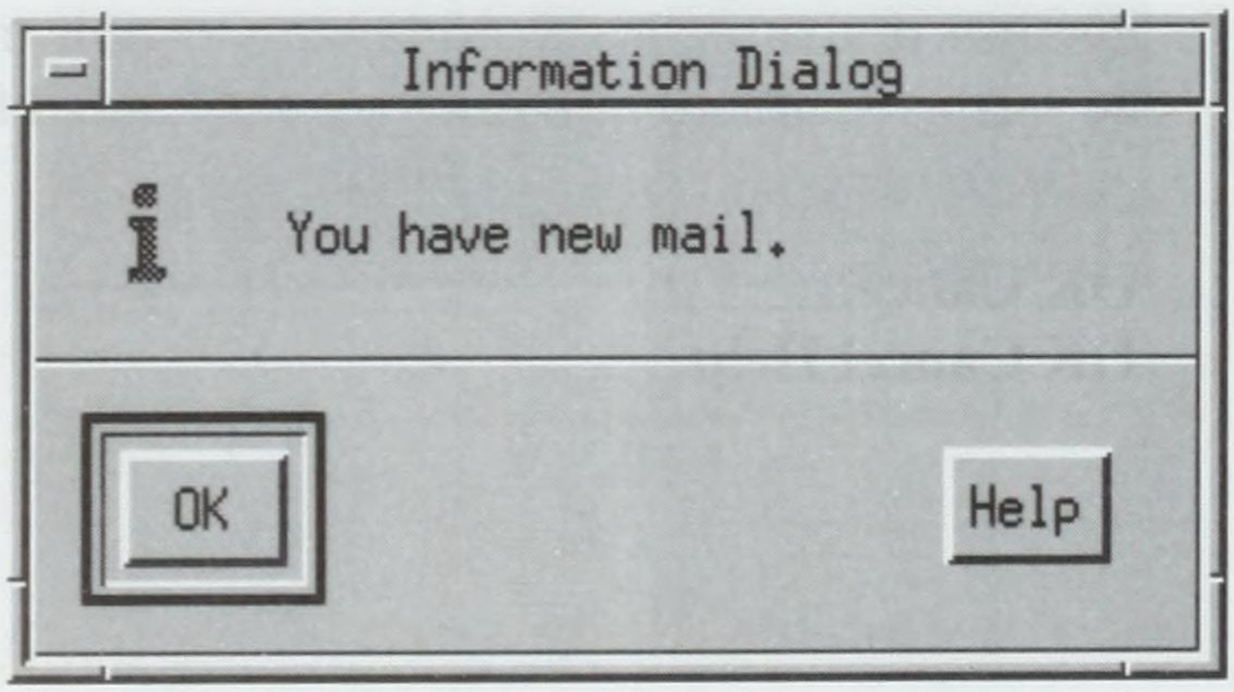
Figure 4-9. An ErrorDialog



An InformationDialog should be used to convey information to the user. It must not interrupt the user's interaction with the * application; that is, it must not be modal. It should include an information symbol, a message, and one of the following button arrangements as shown in Figure 4-10.

OK
OK Help

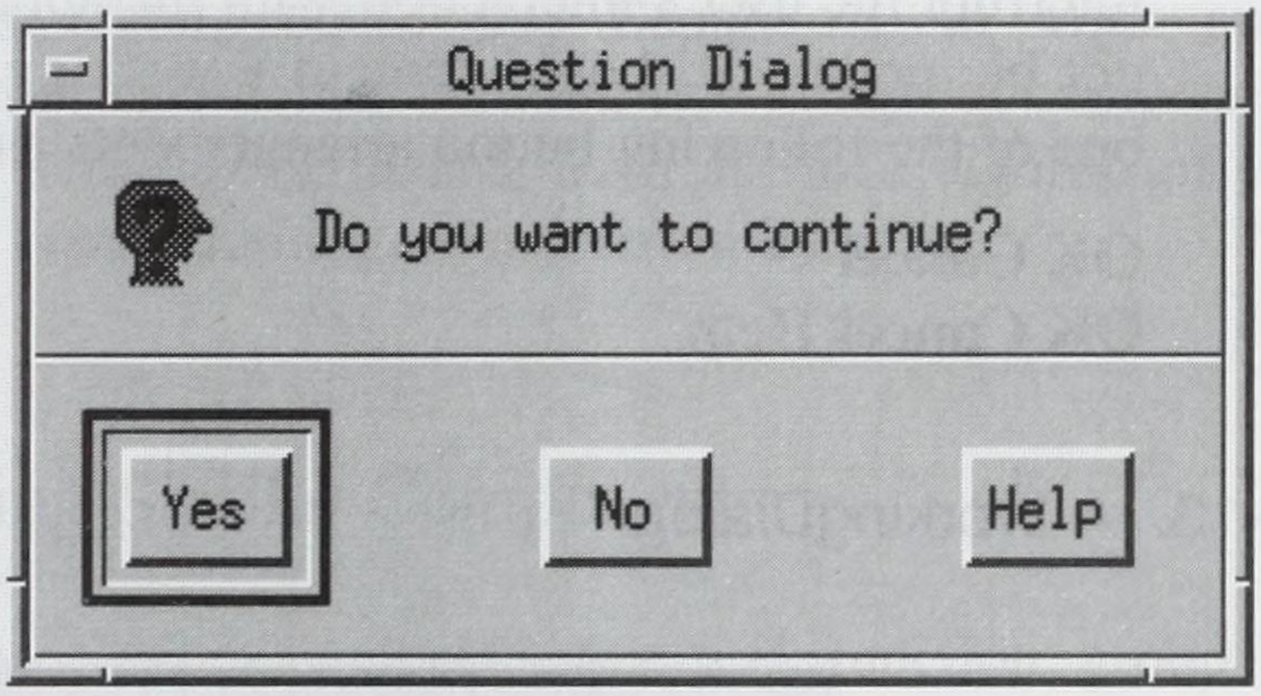
Figure 4-10. An InformationDialog



A QuestionDialog should be used to get a user response to a question. It should interrupt the user's interaction with the application; that is, it should be application modal. It should include a question symbol, a message, and one of the following button arrangements as as shown in Figure 4-11.

Yes No
Yes No Help
Yes No Cancel
Yes No Cancel Help

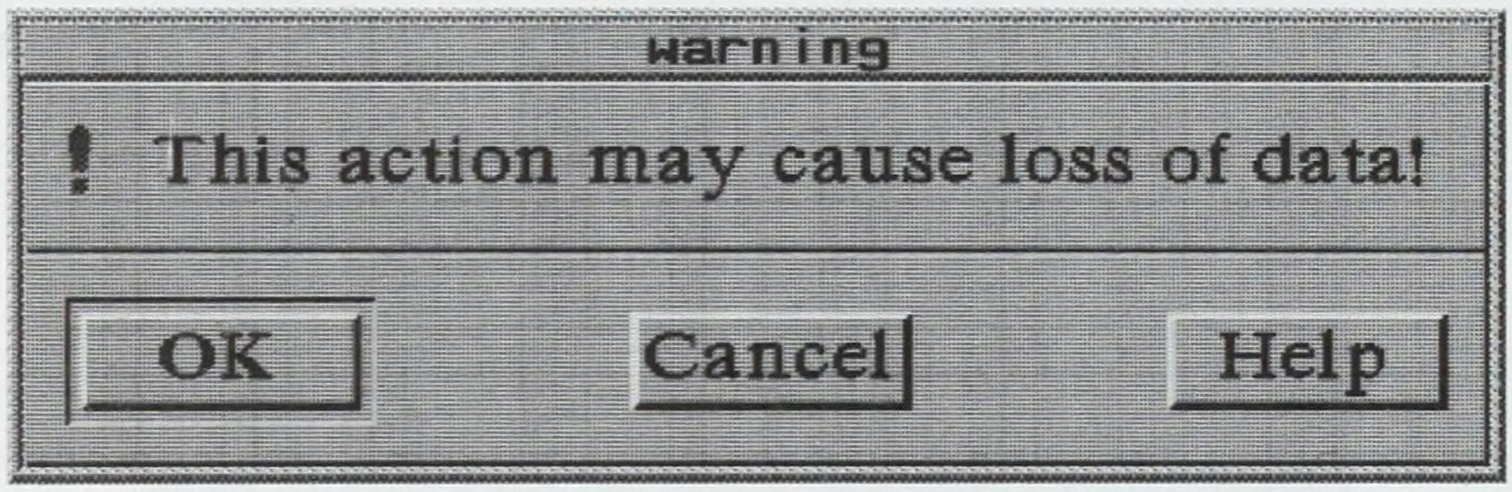
Figure 4-11. A QuestionDialog



A WarningDialog should be used to alert the user to a possible danger. It should interrupt the user's interaction with the application; that is, it should be application modal. It should contain a warning symbol, a message, and one of the following button arrangements as shown in Figure 4-12.

Yes No
Yes No Help
OK Cancel
OK Cancel Help

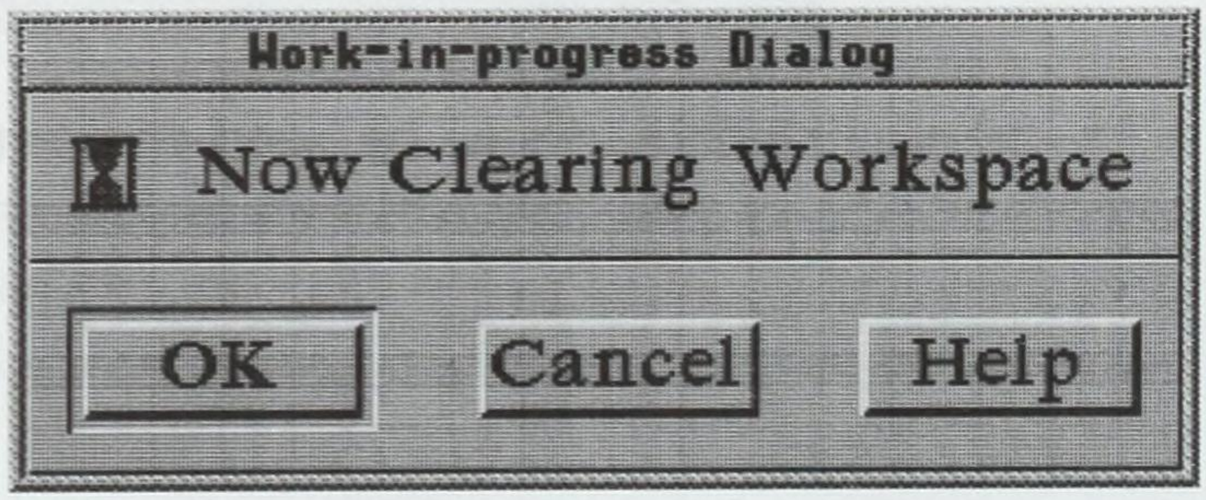
Figure 4-12. A WarningDialog



A WorkingDialog should be used to show work in progress and give the user an opportunity to cancel the operation. It should not interrupt the user's interaction with the application; that is, it should not be modal. It should contain a working symbol, a message, and one of the following button arrangements as shown in Figure 4-13.

OK Cancel
OK Cancel Help

Figure 4-13. A WorkingDialog



4.2.2 Grouping Components

This section describes how you should group components using the group components described in Section 4.1.

- Grouping similar components
- Arranging components for fixed layout
- Arranging components for resizing layout
- Arranging components for aligned layout
- Arranging components in PanedWindows

Separators are a good tool for visually separating groups of components. You should use Separators in your application anyplace where the border between two groups of components is not obvious by some other means, such as for a significant change in component types, framing, or a division by space.

4.2.2.1 Grouping Similar Components

Components similar in appearance and function group naturally together. You should organize similar components together. Similar PushButtons, as in a Menu, and a Panel of CheckButtons, as in a list of options, are good examples of where grouping is important. Without grouping, related actions are difficult to find. One of the most important cases is a Panel of RadioButtons. Without grouping, when you turned on one RadioButton, you might not see the previous button turn off. You can also associate a title with a group of components to better associate the group and its purpose.

4.2.2.2 Arranging Components for Fixed Layout

Components that are grouped using a Composition group component can be positioned as they best fit, both physically and visually. This is often the case in the MainWindow client area, and is also useful when the layout of the components is important to the application.

When you layout components in this manner, remember that your application can be resized by the user. This can be compensated for in a number of ways. If the area has a fixed size and is not appropriate to scroll, like a control panel, you can either clip the area or turn off resizing. If the area has a fixed size and is appropriate to scroll, you should include ScrollBars for scrolling the viewable area.

4.2.2.3 Arranging Components for Resizing Layout

Composition component groups can also be positioned so that the position and size of each component is relative to the Composition component group or other components in the group. This style of layout permits the components to change size proportionally to any change in the size of the Composition component.

Sometimes a window is resized too small to be useful. Your application can either simply ignore this, clip the region once all the components are at their minimum size, remove less useful components to make room for more shrinking, or replace all the components with a message that indicates that the minimum size is reached, stating that the user needs to enlarge the window to continue working in it.

4.2.2.4 Arranging Components for Aligned Layout

Panels can arrange components aligned horizontally, vertically, or in two dimensions. Buttons are usually aligned in Panels horizontally along the bottom of of the client area, either in a MainWindow or a DialogBox. ToggleButtons should be arranged in vertical Panels so the graphics align and look neat. A common use of Panels is in building Menus, or RadioBoxes; that is, a set of RadioButtons. Two-dimensional Panels are commonly used in graphics programs to present drawing styles, and in spreadsheets to contain cells.

Your application should use Panels to provide the best organization for large groups of similar components. Panels should be placed in a Composition component group to allow for proper user resizing either by using ScrollBars or by resizing the elements of the Panel.

4.2.2.5 Arranging Components in PanedWindows

PanedWindows provide a way for the user to simply adjust the size of components relative to one another. You should use PanedWindows to separate user tasks in an application with limited space. This allows the user to ignore elements of the application that are unused.

PanedWindows can also be used to present two simultaneous views of the same data. For example, a text editor may use multiple Panes, with a separate Text component in each Pane to allow the user to write in one section of a document, while looking at another. The user can then resize the Panes to show more or less of either block of text.

PanedWindows can be composed of either vertical (one on top of the other) or horizontal (side by side) sets of Panes, Separators, and Sashes. Users can resize Panes by dragging the boundary between them. Making one Pane bigger makes the other Pane smaller, while the overall size of the window remains the same.

4.2.3 Menu Design

Menus are the primary means of organizing most of an application's features. Because of screen size limitation and visual simplicity, Menus organize both components used frequently by users and components used in most application sessions.

There are three types of Menus.

- PullDown Menus
- Popup Menus
- Option Menus

PullDown Menus are pulled down from a CascadeButton. CascadeButtons should always be available in the context that they are needed. Menus can also contain CascadeButtons, so that Menus can be nested. The MenuBar is a horizontal collection of CascadeButtons.

Popup Menus are context sensitive, but give no cue to their existence. They are popped up when the user presses **BMenu** over a component with an associated Popup Menu. Popup Menus should only be used to provide shortcuts, since new users of an application may not realize or remember that they exist. Even within a single control, such as a Canvas, the contents of a Popup Menu can depend upon position within the control, or the state of the elements at that position, such as whether the Menu is popped up from within a selected range of elements.

Option Menus provide a means of selecting from a set of choices while taking up very little space. An Option Menu is popped up from an OptionButton, which is distinguished by a bar graphic on the right side of the button.

Menus are composed of titles, elements, mnemonics, and accelerators. A Menu's title should be unique to avoid confusion. The title should clearly indicate the purpose of the Menu.

- A PullDown Menu's title is taken from the label in the CascadeButton.
- A Popup Menu's title should be placed at the top of the Popup Menu and separated from the Menu elements by a Separator.
- An Option Menu's title is usually a label to the left of the OptionButton, but can be at the top of the Option Menu itself.

Most basic controls can be Menu elements including Labels, Separators, PushButtons, ToggleButtons, and CascadeButtons. The elements can be identified by either a text label or a graphic. A Menu must be wide enough to accommodate its widest element. *

A mnemonic provides a quick way to access Menu elements from the keyboard. While a Menu is posted, pressing the mnemonic letter of an element activates that element. The MenuBar's mnemonics can be used by pressing **MAlt** with the mnemonic letter. An element's mnemonic should be the first character of the element's label. If that character conflicts with another mnemonic

in the Menu, another character in the label should be used. The mnemonic of an element should be underlined in the element's label. When the appropriate mnemonic letter does not appear in the element's label, it should appear in parentheses after the label.

An accelerator provides a way to access Menu elements from the keyboard without posting the Menu. Accelerators are useful to the experienced user for saving time when using frequently used components. You should provide accelerators primarily as a matter of utility, not design conformity.

If a keyboard accelerator exists for a Menu entry, it should appear following the Menu's label, justified on the same line. The accelerator and the selection should be separated by enough space to make them visually distinct.

You should use the following guidelines when designing Menus and Menu systems:

- Keep Menu structures simple.
- Group like Menu elements together.
- List Menu selections by frequency of use.
- List Menu selections by order of use.
- Separate destructive actions.
- Provide mnemonics and accelerators.

4.2.3.1 Keeping Menu Structures Simple

Applications should keep Menu structures simple. One of the primary benefits of Menus is the ease of access to the elements of the Menu. While cascading submenus help the application and the user organize Menu elements, each level of a submenu reduces the ease of access to the Menu elements. Multiple levels of cascading submenus can also quickly create visual clutter. Whenever you consider using a cascading submenu, you should consider using a DialogBox or more PullDown Menus instead.

4.2.3.2 Grouping Like Menu Elements Together

Applications should group Menu elements into logical groups. This helps the user locate specific Menu elements. You should first try to place a new Menu element into the common Menu groups described in Section 4.2.1.5. If that is not appropriate, you should group new Menu elements according to function, with the more frequently used element appearing first. You should also use Separators between logical groups of elements.

4.2.3.3 Listing Menu Selections by Frequency of Use

As in other client areas, applications should order Menu elements according to the frequency of usage, positioning the most frequently used elements near the top of the Menu.

4.2.3.4 Listing Menu Selections by Order of Use

More important to Menu design than the frequency of use is the order of use. Applications should order Menu elements according to the order of usage. For example, the **Copy** element should be placed before **Paste.** This helps the user's interactions flow smoothly.

4.2.3.5 Separating Destructive Actions

Applications should separate destructive actions from frequently chosen selections. This is to avoid accidental selection of the destructive element. Destructive elements, like **Delete** or **Clear**, should be placed at the end of a PullDown Menu and separated from other elements by a Separator.

4.2.3.6 Providing Mnemonics and Accelerators

Applications should provide mnemonics and accelerators to Menu elements. Try to choose mnemonics and accelerators that are easy to remember by using letters from the element's title. Note that mnemonics and accelerators only add to the utility of your applications. They never detract from the basic ability of a new user.

Applications should provide accelerators for frequently used menu items. In general, accelerators should not be assigned for every menu item in an application. It is preferable to assign accelerators that have some mnemonic value, although accelerators that use function keys are acceptable.

Applications should not use accelerators that are a combination of the modifier **MAlt** and letter keys to avoid conflicts with mnemonics. For example, **MAlt <E>** as an accelerator for **Exit** conflicts with the use of **MAlt <E>** to pull down the Edit menu since it is the mnemonic for the **Edit** CascadeButton in the MenuBar.

Similarly, applications that involve text entry should not use accelerators that are combinations of the modifier **MShift** and letter keys to avoid conflict with the text entry commands. Applications that expect field controls to have bindings that include combinations of the modifier **MCtrl** and letter keys, such as text editors, should also avoid these combinations.

Accelerator bindings that use only one modifier are preferable to bindings that use two or more modifier keys.

4.2.4 DialogBox Design

When designing a DialogBox, you should follow all the same layout principles as for other applications areas. DialogBoxes are usually transitory. The user usually wants to simply respond and get back to the primary tasks of the application. To help the user respond quickly, there are a number of common DialogBox actions. By using and ordering these actions consistently, you provide the user with cues to quickly respond to each DialogBox.

When a DialogBox is displayed, all components within the DialogBox should reflect the current state of the application. For example, if the DialogBox is used for changing the current font in a text editor, the DialogBox should be initially displayed with the current font. If the DialogBox is modeless, then any changes to the application should be updated in the DialogBox.

DialogBoxes may limit how a user can interact with other windows in order to force the order of interaction. These limitations, which are called modes, are described in the following text.

Modeless	Allows interaction with the secondary window and all other windows.
Primary modal	Does not allow interaction with any ancestor of the window.
Application modal	Does not allow interaction with any window created by the same application even if the application has multiple primary windows.
System modal	Does not allow interaction with any window on the screen. This includes windows from all other applications and any icon box. To indicate a system modal secondary window, the pointer should change shape to a caution pointer whenever it leaves the system modal secondary window.

Remember that the guidelines presented in this section, like all the guidelines in this *Style Guide*, only apply in a left-to-right language direction environment in an English locale. You need to make the appropriate changes for other locales.

4.2.4.1 Common Dialog Box Actions

While your application may sometimes require special DialogBox actions, most share common actions. The common actions provide a consistent means for the user to quickly respond to DialogBoxes and get back to the primary application tasks. The common actions

should be presented in a horizontal collection of PushButtons at the bottom of the DialogBox, separated from the rest of the DialogBox by a Separator.

No DialogBox will contain all of the common actions in the following list. You should use the ones appropriate to your application or determine new actions so they do not conflict with the common actions listed. If you create a new action, you should give it an active-voice label that indicates its purpose. A active-voice label describes the action that pressing the button causes. The actions are listed in the approximate sequence in which they should appear in DialogBoxes.

Yes Must indicate an affirmative response to a question posed in the DialogBox, and then close the window. While **Yes** is not an active-voice label, it implies a positive response to a question in a **QuestionDialog** or a **WarningDialog**. Only use **Yes** if it is a clear answer to the question. *

No Must indicate a negative response to a question posed in the DialogBox, and then close the window. While **No** is not an active-voice label, it implies a negative response to a question in a **QuestionDialog**. Only use **No** if it is a clear answer to the question. *

OK Must cause the application to apply any changes made in the DialogBox and dismiss the DialogBox. While **OK** is not an active-voice label, its usage is too common to change. *

Apply Must apply any changes made to components in the DialogBox. *

Retry Must cause the task in progress to be attempted again. This action is commonly found in message boxes that report an error. *

Stop Must end the task in progress at the next possible breaking point. This action is commonly found in a **WorkingDialogBox**. *

Reset Must cancel any user changes that have not been applied to the application. It must also reset the status of the DialogBox to the state since the last time the DialogBox action was applied or to the initial state of the DialogBox. * * *

Cancel Must close the DialogBox without performing any DialogBox actions not yet applied to the application. Pressing **KCancel** anywhere in the DialogBox, except during a cancelable drag operation, must perform the action of this button. *

Help . . . Must provide any help for the DialogBox. The ellipses should not be used if the help action is not provided in a new DialogBox. *

If a DialogBox action causes an error, the DialogBox should not be dismissed before the error is displayed. Instead, the DialogBox should remain available after the error is dismissed to give the user a chance to correct the error and reuse the DialogBox.

4.2.4.2 Arranging Common Actions

You should arrange PushButton actions in DialogBoxes in the same way you arrange other PushButtons, according to order and frequency of use. The common action PushButtons should be ordered as presented in the previous section. Positive responses to the DialogBox should be presented first, followed by negative responses, and canceling responses. **Help** should always be the last action on the right.

The following rules should be used when determining what default buttons to place in a DialogBox.

- Modal DialogBoxes should use one of the following button arrangements unless superseded by another rule:

  OK Cancel
  OK Cancel Help

- Modeless DialogBoxes should use one of the following button arrangements unless superseded by another rule:

  OK Apply Cancel
  OK Apply Cancel Help
  OK Apply Reset Cancel
  OK Apply Reset Cancel Help

- Information DialogBoxes should use one of the following button arrangements:

  OK
  OK Help

- Question DialogBoxes should use one of the following button arrangements:

  Yes No
  Yes No Help

  It is possible that both the **Yes** and **No** actions of a Question DialogBox will perform an action. If this is the case, the Question DialogBox should use one of the following button arrangements:

  Yes No Cancel
  Yes No Cancel Help

- Warning DialogBoxes should use one of the following button arrangements:

  Yes No
  Yes No Help
  OK Cancel
  OK Cancel Help

- Working DialogBoxes should use one of the following button arrangements:

  Cancel
  Cancel Help
  Retry Cancel
  Retry Cancel Help

4.2.4.3 Determining DialogBox Location and Size

Your application determines the size and location of its DialogBoxes. You should size and place DialogBoxes so that they do not obscure important information in other windows of your application. The initial size of a DialogBox should be large enough

to contain the dialog components without crowding or visual confusion, but otherwise should be as small as possible. DialogBoxes should follow the same rules for resizing as a MainWindow as described in Section 4.2.2.3.

You should place DialogBoxes on the screen so they are completely visible.

In general you should place DialogBoxes close to either the component that caused it to be displayed, the current action, or the information needed to respond to it. When a DialogBox relates to an item in an underlying window, you should position the DialogBox to the right of the item. If there is not enough room to the right of the item, try to position the DialogBox to the left, below or above the item, in that order, depending on screen space available. You should only obscure related information as a last resort.

If a DialogBox does not relate to items in the underlying windows, the DialogBox should be placed centered in the application's work area.

If two DialogBoxes need to overlay, you should offset the top DialogBox to the right and below the title of the lower DialogBox. Use your best judgement, knowing that the screen area for DialogBoxes is limited.

While the previous suggestions seem simple enough, they cannot always be followed completely. Therefore, DialogBoxes, once displayed, should be movable so that the user can relocate them as needed to see information in underlying windows.

4.3 Interaction

This section gives guidelines for creating applications with consistent interactions. When an application behaves as expected and the user is not surprised by the results of the actions, the user can complete tasks quicker. This section presents the following guidelines for good application interaction:

- Supply indications of actions.
- Give feedback.
- Allow user flexibility.

4.3.1 Supplying Indications of Actions

The first step to consistent interaction is to provide cues to the result of every action. This means that actions of components should be indicated by the component's shape, label, and graphics. It also means that the actions and interactions of components should remain consistent, so the user always knows what to expect.

Lastly, it means that interactions should be simple. As interactions become complicated, it also becomes difficult to visually represent the interaction. Complicated interactions and components create the possibility for more errors. Even the most complicated concepts can be clarified by careful organization, so if your application's interactions seem complicated, consider reorganizing them for simplicity.

4.3.1.1 Using Common Components

Users expect components to behave consistently across all applications. PushButtons always perform an action. OptionButtons always provide selections. Because of this, when users want to perform an action, they look for a PushButton, usually in a Menu. They do not look for an OptionButton. You should use the

components that are provided when appropriate, rather than create new ones. You should not alter the look of a component so that it is unrecognizable as the type of component that it is.

4.3.1.2 Using Intuitive Labels

One of the best indicators of the action of a component is its label. The label can be either text or a graphic. You should choose your labels carefully to indicate the action of each component.

Components that perform actions should be labeled with active verbs. Components that present options should be labeled with nouns. You should also label component groups, including Panels, with nouns to indicate the the contents of the group.

Consider the use of graphics as labels with two cautions. Graphic labels cannot be nouns or active verbs, so choose a graphic whose meaning is clear in the context of the component. Also remember that while graphic symbols are very language independent, they can be highly culture specific. In some cultures, a mailbox graphic can indicate a mailer action, but since mailboxes are not common in all cultures, an envelope graphic may be better, or you can provide a mechanism for changing the graphic based on the locale.

4.3.1.3 Using Graphics to Show Action

Many components also include a small graphic symbol following the label to indicate the action of the component. CascadeButtons should use an arrow graphic that points in the direction the cascading Menu will appear. OptionButtons should use a rectangle graphic to distinguish them from PushButtons. Any component that causes a DialogBox to be displayed should include an ellipses following the label.

All previously mentioned graphics should follow the text or graphic of the label. In left-to-right direction language environments, the graphic should be on the right. In right-to-left direction language

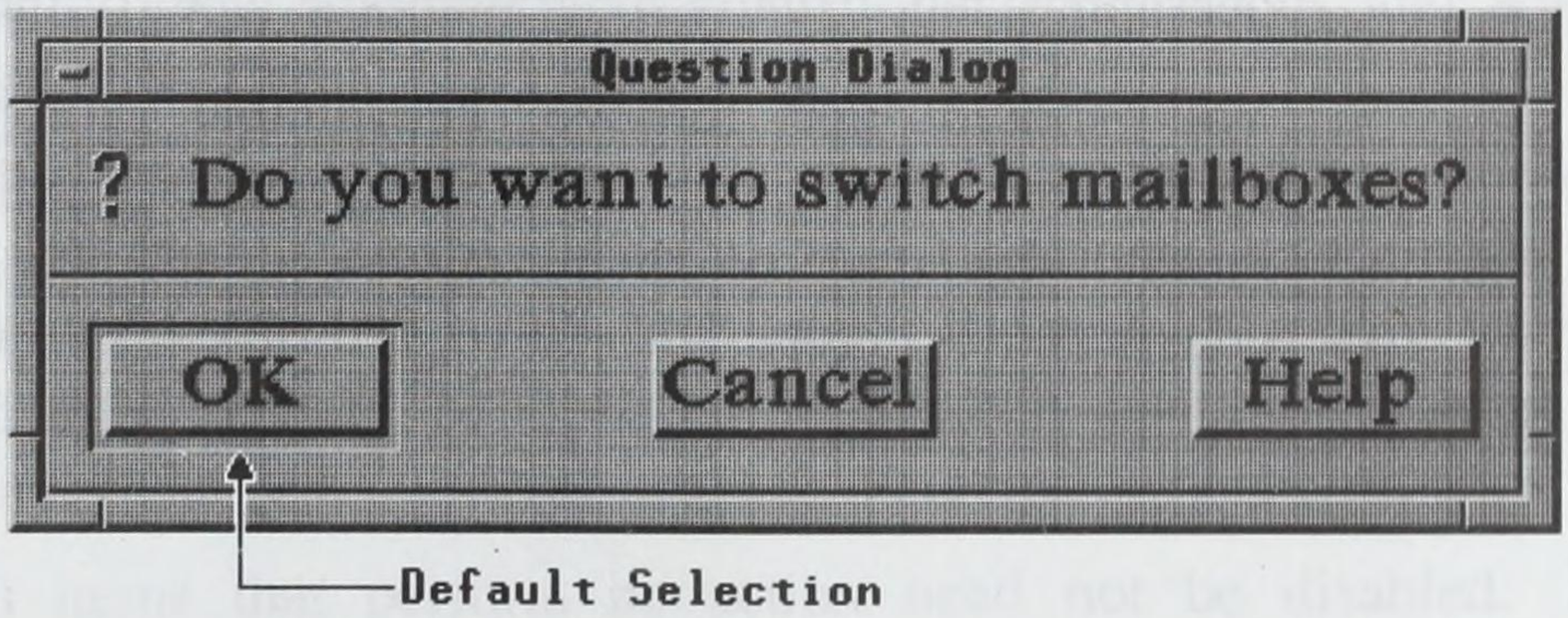
environments, the graphic should be on the left. Note that the graphic indicating the state of a ToggleButton precedes the label in a left-to-right direction language environment.

4.3.1.4 Showing Default Actions

Your application should use default values for common settings or obvious selections. Default values should be shown in the on state. For example, the default value for a Text area should be in the Text area in the selected state whenever text entry is requested; the default selection in a List should be set in the selected state whenever a list selection is requested; or the default RadioButton should be filled in a Panel at application start-up time. In any case, once the state is changed, the new state should take the place of the default until the state is reset. Your application can decide whether to save its state after being closed.

Groups of controls, such as a DialogBox, can also have a default action. The default action is usually activated by pressing **KActivate. KActivate** is usually bound to **<Return>** or **MCtrl <Return>**. The default action of a component group should be distinguished from the other selections by an extra border as shown in Figure 4-14.

Figure 4-14. A Default PushButton



The **OK** PushButton should be the default PushButton in modal DialogBoxes and modeless DialogBoxes that are transient. The **Apply** PushButton should be the default PushButton in a modeless DialogBox that is likely to be displayed for multiple actions.

Using the keyboard to navigate through the PushButtons, the button with the location cursor should become the default PushButton. This ensures that pressing **KActivate** over a PushButton invokes the correct PushButton. When the location cursor leaves the PushButtons, the original default button should once again become the default.

4.3.1.5 Avoiding Component Modes

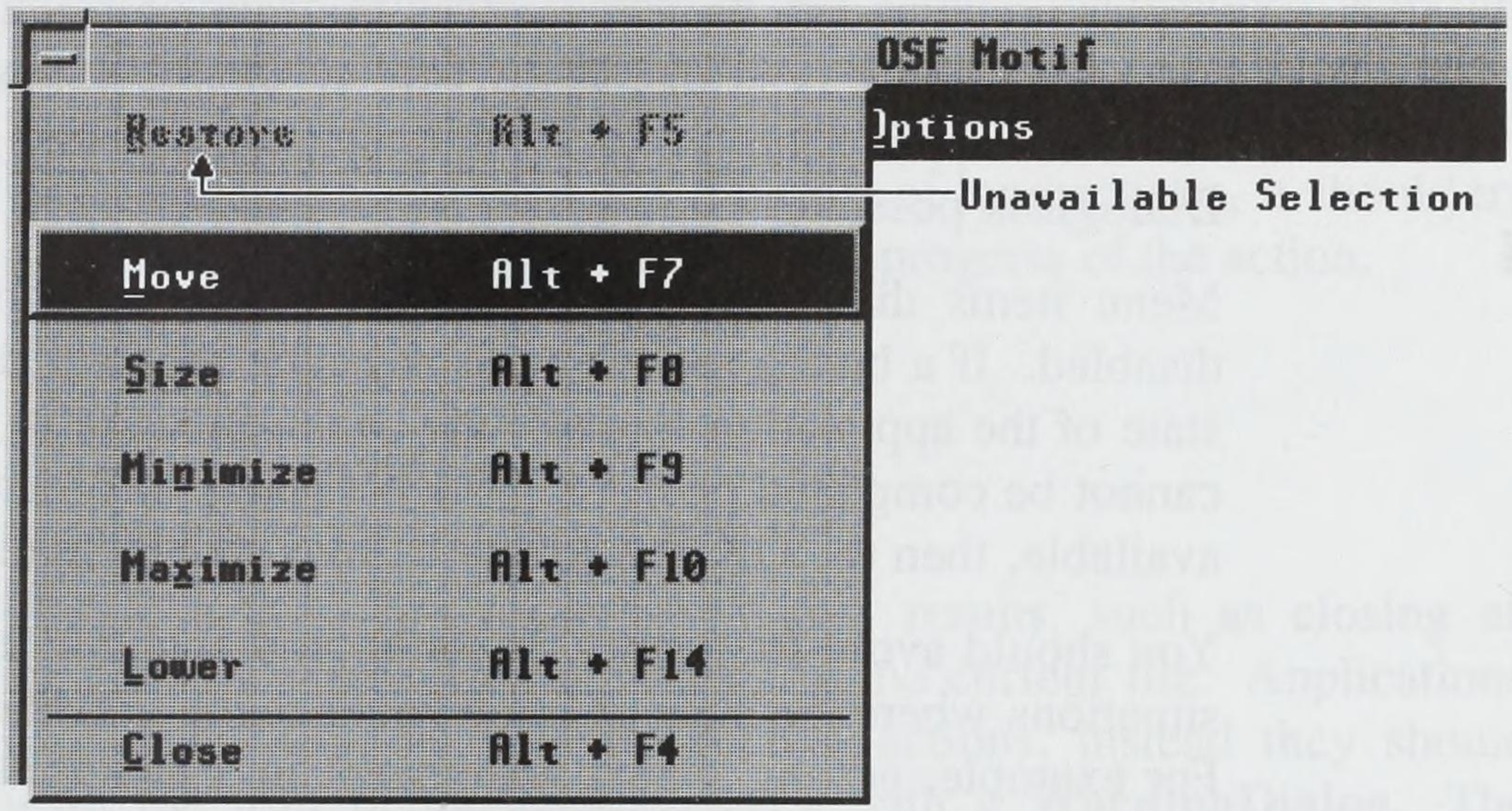
A component has modes when its action changes based on some previous action or the state of the application. This is very confusing to the user, who was expecting the original action of the component. Components in your application should not have modes. Your application should use multiple components, rather than modal components.

4.3.1.6 Showing Unavailable Components

As the state of your application changes, certain components become inappropriate. For example, the **Minimize** selection in a window Menu is inappropriate when the window is already minimized. In such cases, you should make the inappropriate components unavailable. This is also called disabling the components. Disabled components should be visually deemphasized, usually by graying the label of the component.

You should not remove unavailable components from the application client areas. The components should remain visible to remind the user of their existence, and to ensure application consistency. Figure 4-15 shows a disabled Menu element.

Figure 4-15. A Disabled Menu Element



You should disable Menu items and components to help avoid errors. In general, you should disable the lowest level component that results in an irreversible error state. For example, consider a graphics editor that has a DialogBox that is used for aligning selected graphics objects. If the user may want to see the choices in the DialogBox even if the choices are not available, this DialogBox should be displayable at all times; its Menu item should not be disabled. If there are no graphics objects to align, the DialogBox should still be displayed, but its **OK** and **Apply** buttons should be disabled. Once graphics objects are selected, the **OK** and **Apply** buttons should become enabled.

Menu items that are inappropriate and that will result in error messages should be disabled. Some examples are the **Edit** Menu's **Cut** and **Copy** actions when nothing is selected. The **Edit** Menu's **Undo** entry should be disabled if the last operation cannot be undone.

Menu items that perform no action need not be disabled. For example, the **New** entry in the **File** Menu need not be disabled immediately after it is invoked. It can be selectable repeatedly, even though it performs no new action, because it does not result in an error state.

A Menu item that displays a modal DialogBox should be disabled if it would cause an error either on display of the box, or on clicking the **OK** or **Apply** buttons in the DialogBox. Since the DialogBox is modal, nothing can be changed in the application to correct the error until the box is dismissed, and it should not be displayable until the application is in a state that makes the action of the DialogBox possible.

Menu items that display modeless DialogBoxes should never be disabled. If a box is modeless, at any time the user can change the state of the application to make the DialogBox useful. If the action cannot be completed because some necessary information is not yet available, then the **OK** and **Apply** buttons should be disabled.

You should avoid frequently disabling and enabling components in situations where the state change would cause a distracting flashing. For example, editing a Text component in a DialogBox may cause some buttons to be inappropriate at each invalid text value. In this case, you should display an error message if the **OK** or **Apply** button is clicked for an inappropriate text value. Of course, the error message should explain the valid text values.

4.3.2 Providing Feedback

Another important element to user interaction is providing feedback about the current state of the application. This is done, as described in the previous section, using labels and graphics, and by keeping the interface consistent. Your application should also dynamically indicate the state of the application's actions. For example, the mouse pointer shape changes to indicate when and where special actions can occur. Chapter 2 describes mouse pointer shapes in detail. Other ways to provide the user feedback are described in the following sections.

4.3.2.1 Showing Progress

If an action takes a long time to complete, the user may mistake the delay to mean that the system or the application stopped working. For actions that take a long time to complete, your application should indicate that there will be a delay with a **WorkingDialog**. If your application can track the progress of long actions, it should try to update the **WorkingDialog** with the progress of the action.

4.3.2.2 Providing Warnings

Certain actions can cause destructive results, such as closing an application before saving changes in the current file. Applications should not disallow such destructive actions, instead they should warn the user of the consequences with a **WarningDialog.** The * **WarningDialog** must allow the user to cancel the destructive * action. Note that too many **WarningDialogs** can be disruptive to the user's main task. **WarningDialogs** should be reserved for truly destructive actions. For destructive actions that can easily be recovered from, applications should provide undo actions to reverse them; for example, the **Undo** element of the **Edit** PullDown Menu.

4.3.2.3 Providing Help

Even in the most intuitive application, the purpose of a component or the way to do a task can be hard to figure out for a new user. Your application should provide a help mechanism for all of its aspects. Section 4.2.1.5 describes the most common base for a help mechanism, the **Help** PullDown Menu on the MenuBar. Context-sensitive help should also be available by using the **KHelp** key. **KHelp** is usually bound to **<F1>** or **<Help>**.

4.3.3 Allowing User Flexibility

Good user-application interaction should also allow user flexibility. No matter how well your application is designed, some users will not like parts of it. They will want to change some elements of it; for example, from simple elements like the colors and fonts to complicated elements like the default values. You should allow users to adjust elements of your applications because it increases their sense of control over the applications. You should consider the following attributes of your applications for user customization:

- Application parameters
- Colors
- Fonts
- Default values
- Key bindings
- Labels
- Messages
- Help information

The exact list of attributes you should allow the user to customize depends on your application.

4.4 Component Design

For consistency with other applications, you should always try to use existing components for your application tasks, but there are cases where new components are needed. In designing new components, you should follow the same rules as application designers follow. You should think of components as small applications. They perform a task, present information to the user, and take information from the user. New components must follow * the guidelines for designing applications.

The first step to designing a new component is to compare its features with those of the other components. If the new component has a feature that is the same as another component, the mechanisms for using the feature, layout, key bindings, graphics, and so on, should be the same in the new component. You should also try to match the appearance style of components on your system. Most OSF/Motif compliant systems use a 3-dimensional beveled presentation style. Appearance is not specified as a matter of component style in this *Style Guide*; however, any new components will assimilate better with existing components if they are designed to conform to the implied appearance style for the system on which they will be used.

Chapter 5

Window Manager Design Principles

A window manager is a specialized application. In designing a window manager you must follow the same principles as for any other application. A window manager must also follow the style guidelines for input, navigation, and selection models as set forth in Chapters 2 and 3.

There are a few elements of user interface design that are specific to window managers. This chapter discusses those elements.

- Configurability
- Window support
- Window decorations
- Window placement
- Window navigation
- Icons

5.1 Configurability

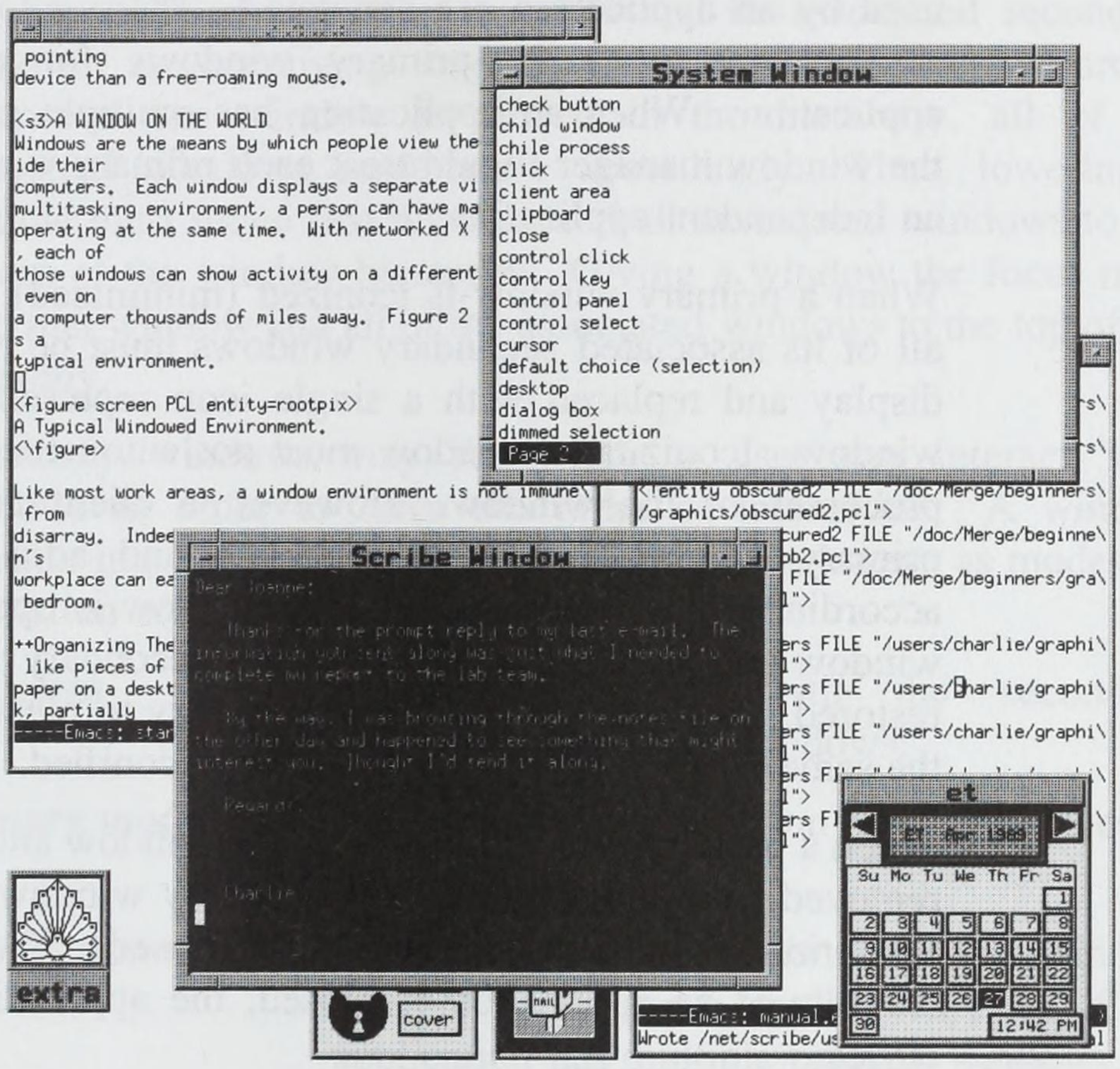
This chapter describes only those elements of window manager design that help create a consistent user interface. A window manager can supply a mechanism that allows the user to configure the window manager. The window manager can make any element user configurable, including key bindings, Menu contents, default window decorations, or any other rules and element defined in this guide. Any window manager that supplies a configuration * mechanism must also include support for toggling between the * current user configuration and the default configuration as mandated * by this guide. The configuration toggle key press, **MAlt MShift MCtrl !**, must initiate a configuration toggle. Any request for toggling the configuration should request user verification before proceeding.

5.2 Window Support

Users communicate with applications using windows. A window is an area of the screen (usually rectangular) that provides the user with the functional means to communicate with an application and through which an application can communicate with the user.

A typical environment has several applications in operation simultaneously. Each application typically has a main or primary window that displays data and in which the user carries on primary interaction with the application. Applications can have additional windows to communicate context-specific interactions with the user of the application. These additional windows are called secondary windows, or transient windows. DialogBoxes are often used to create secondary windows. Figure 5-1 illustrates a typical OSF/Motif environment.

Figure 5-1. A Typical OSF/Motif User Environment



While each application can be made up of many windows, each window is one of only three basic types:

- A primary window, the main application window.
- A secondary window, a window that provides secondary and transient interaction with the user.
- A menu window, a temporary window for displaying a menu of choices for the user.

A window manager must support multiple applications, each with one or more primary windows. The window manager must also recognize and support secondary windows. Also, it must associate each secondary window with a primary window or another secondary window.

5.2.1 Primary Window

A primary window is the window from which all the other windows used by an application are generated. The window manager must * support one or more primary windows for each windowing * application. When an application has multiple primary windows, the window manager should treat each primary window as if it were an independent application.

When a primary window is iconized (minimized), the window and * all of its associated secondary windows must be removed from the * display and replaced with a single icon representing the primary * window. Iconizing a window must not automatically suspend any * processes of the window. However, a client can notice that a primary window has been unmaped and adjust its processing accordingly. When the icon of the window is opened, the primary * window and all the current associated secondary windows must be * restored. The window manager should try to place the windows in the same position from which they were iconified.

When a primary window is closed, the window and its icon must be * removed from the display. All secondary windows associated with * the primary window must also be closed. If the last primary * window of an application is closed, the application must also be * closed.

5.2.2 Secondary Windows (Dialog)

Applications use secondary windows to conduct context-specific dialog with the user. Such context-specific dialogs are usually transitory, and the secondary windows can be removed when they are no longer needed. When a secondary window is closed, its * parent must not be affected, and any secondary windows that are * children of it must also be closed.

Secondary windows are always related to a parent window. Sometimes the parent is a primary window, sometimes another secondary window. Any window can have any number of secondary window children.

Secondary windows are not constrained to be clipped within their parent window, but they must always appear on top of that parent window in the window hierarchy. In a layered window manager, you can think of a primary window and its associated secondary windows as occupying one layer in the window hierarchy. Whenever one window is moved in the hierarchy, all of the associated windows must move accordingly. Thus, lowering a window must move that window and all associated windows to the bottom of the window hierarchy: Giving a window the focus must raise that window and all of the associated windows to the top of the hierarchy.

Secondary windows may limit how a user can interact with windows in order to force the order of interaction. A window manager must support the following four types, known as modes, of interaction with secondary windows.

Modeless	Allows interaction with the secondary window and all other windows.
Primary modal	Does not allow interaction with any ancestor of the window.
Application modal	Does not allow interaction with any window created by the same application even if the application has multiple primary windows.
System modal	Does not allow interaction with any window on the screen. This includes windows from all other applications and any icon box. To indicate a system modal secondary window, the pointer should change shape to a caution pointer whenever it leaves the system modal secondary window.

A window manager must not allow focus to be given to any window that is not allowed to accept input because of the modality of a DialogBox.

5.2.3 Menu Windows

Menu windows are used to present Menus. They are not specifically created by applications, but by the components used to create the application. Menu windows are always related to a parent window. The parent can be either a primary window, secondary window, or another menu window. A window can only have one menu window child at a time.

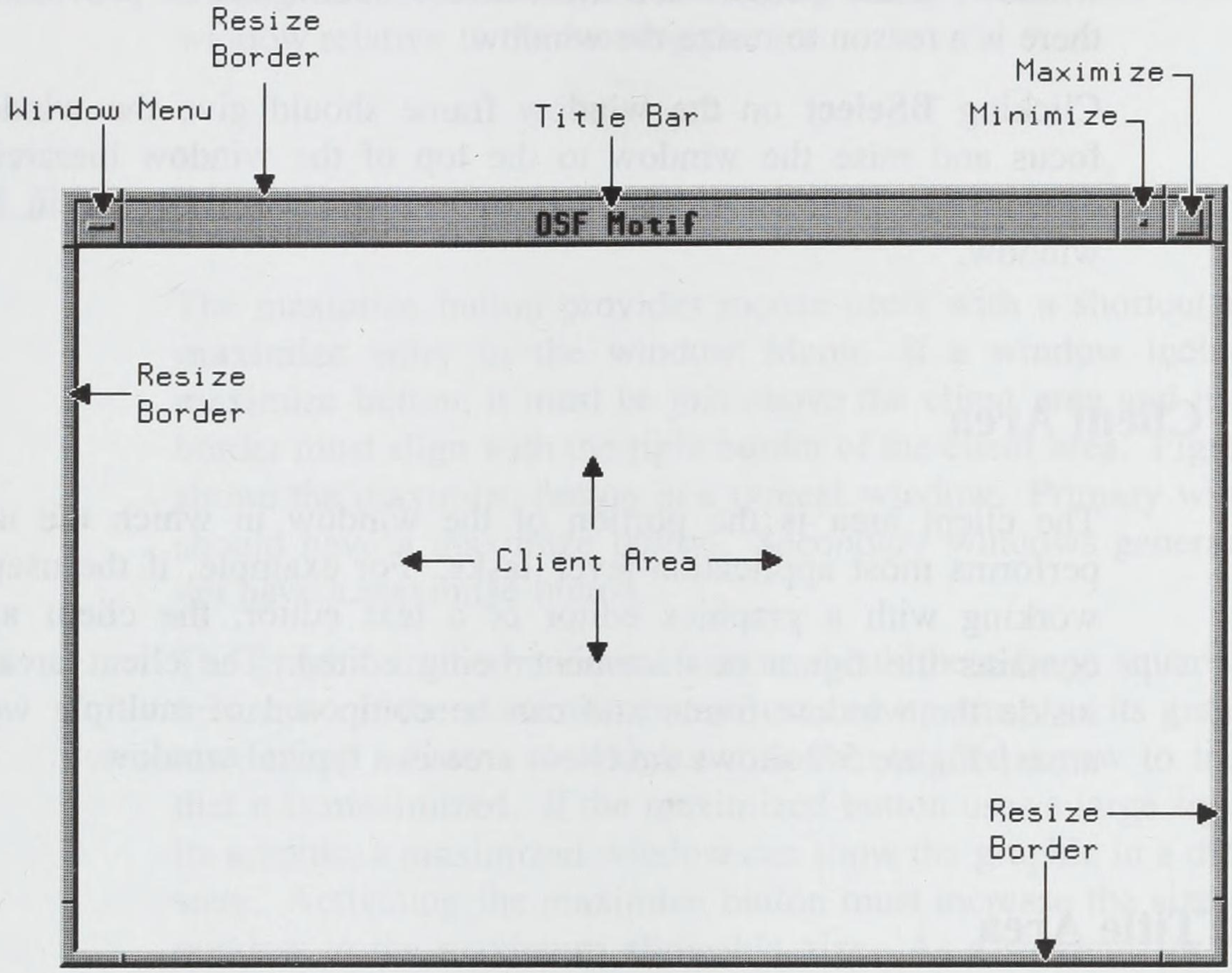
A menu window is very short lived. It is only available to a client while no mouse or button actions are being performed elsewhere on the screen. Once interaction starts in another window on the screen, * except a child menu window, the menu window must be removed. Some systems have menu windows, called Tear-off Menus, that can be made to remain on the screen during other interactions. Once * these menus are made permanent, they must become secondary * windows.

Menu windows are not constrained to be clipped within their parent window, but they must always appear on top of that parent window * in the window hierarchy. The window manager must not supply any * window decoration to menu windows.

5.3 Window Decorations

A window manager can provide windows with a window frame that contains components, often called decorations. The window decorations allow user interaction with the window manager. Along with the frame components, a window manager contains a client area. The client area is the display area for an application. Figure 5-2 shows a typical window and its decorations.

Figure 5-2. A Typical OSF/Motif Window Layout



A window manager can support any number of window decorations, but must support the client area and the window frame. A window manager must support the following window frame components:

- Title area
- Maximize button
- Minimize button
- Resize border
- Window Menu button

Each window must have a client area. A window manager must allow each window to choose which combination of decorations to include, except that a window must not have any buttons without a title area. A primary window must by default have all the decorations: window Menu button, maximize button, minimize

button, title area and resize border. A secondary window must by * default have a window Menu button, and title area. In a secondary * window, resize borders and a maximize button can be provided if * there is a reason to resize the window.

Clicking **BSelect** on the window frame should give the window focus and raise the window to the top of the window hierarchy, except when dialog modality disallows focus to be set in the window.

5.3.1 Client Area

The client area is the portion of the window in which the user performs most application-level tasks. For example, if the user is working with a graphics editor or a text editor, the client area contains the figure or document being edited. The client area is inside the window frame and can be composed of multiple work areas. Figure 5-2 shows the client area in a typical window.

5.3.2 Title Area

The title area, also called the title bar, supplies a place to identify the window. If a window includes a title area, it must be a * horizontal bar at the top of the window. It must be just above the * client area and lie horizontally between the window Menu (or other * buttons) on the left and the window control buttons on the right. Figure 5-2 shows the title area in a typical window.

The title area should contain a short string called the title that labels the contents of the window. The title must be settable at window * startup both by the user and by the application. For applications with a single primary window, the title should be the name of the application. For applications with multiple primary windows, the title should indicate the purpose of the window. The title in secondary windows should indicate the purpose of the window. The title can also contain other useful information, such as the machine on which the application is running, the current directory, or similar relatively static information.

Besides supplying a location for a label, the title area also acts as a position, or handle, for moving a window. Pressing **BSelect** or * **BDrag** in the title area and dragging the pointer must move the * window relative to the moving pointer.

5.3.3 Maximize Button

The maximize button provides mouse users with a shortcut to the **maximize** entry in the window Menu. If a window includes a * maximize button, it must be just above the client area and its right * border must align with the right border of the client area. Figure 5-2 shows the maximize button in a typical window. Primary windows should have a maximize button. Secondary windows generally do not have a maximize button.

The graphic in the maximize button should be a large square or an up arrow. If the maximize button uses an up arrow as its graphic, a maximized window should use a double-headed arrow to indicate that it is maximized. If the maximized button uses a large square as its graphic, a maximized window can show the graphic in a different state. Activating the maximize button must increase the size of the * window to the maximum allowable size. As a shortcut for mouse * users to the **Restore** entry in the window Menu, activating the * maximize button of a maximized window must restore the window * to its size and location before being maximized.

5.3.4 Minimize Button

The minimize button provides mouse users a shortcut to the **Minimize** entry in the window Menu. If a window includes a * minimize button, it must be just above the client area and directly to * the left of the maximize button. Figure 5-2 shows the minimize button in a typical window. Primary windows should have a minimize button. Secondary windows must not have a minimize * button.

The graphic in the minimize button should be a small square or a down arrow. Activating the minimize button must iconify the window unless the window family contains a system modal DialogBox. *

5.3.5 Other Buttons

You can bind additional window manager functions to buttons on the window frame. Any additional buttons must be placed directly to the left of the minimize button or directly to the right of the window Menu button and above the client area. Each button action must correspond to a entry in the window Menu. *

5.3.6 Resize Borders

Applications may suggest the initial size of their windows to the window manager. Window sizes can vary according to the work performed in them. At any time, a user should be able to alter the size of most windows. The **Size** entry in the window Menu provides a method for the user to alter the size of windows. The resize borders provide a shortcut for mouse users for the **Size** entry in the window Menu. Resize borders are not generally provided on secondary windows.

The resize borders are the outermost components of the window manager frame. They are made up of two components: the corner handles and the edge handles. If a window includes resize borders, there must be one corner handle in each corner of the window at its extremes, and one edge handle between each pair of corner handles. There must be no window components outside the boundary formed by the resize borders. Figure 5-2 shows resize borders in a typical window. *

Pressing **BSelect** or **BDrag** in a corner handle and dragging the * pointer must change the height and width of the window relative to * the moving pointer without changing the position of the opposite * corner. Pressing **BSelect** or **BDrag** in a top or bottom edge handle * and dragging the pointer must change the height of the window * relative to the moving pointer without changing the width or the * position of the opposite edge. Pressing **BSelect** or **BDrag** in a side * edge handle and dragging the pointer must change the width of the * window relative to the moving pointer without changing the height * or the position of the opposite edge.

5.3.7 Window Menu

The window Menu, sometimes called the system Menu or control Menu, is used to display the list of window actions. All actions possible for a window should be displayed in the window Menu because keyboard-only users interact with the window manager through this Menu. Because of this, it is rare that a window does not need a window Menu. The window Menu can be configured out by the application or by the user, but should not be removed by the window manager.

If a window includes a window Menu, the window Menu button * must be located just above the client area, the left edge of the button * must align with the left edge of the client area, and the button must * be just to the left of the title area, unless other buttons are included * between the window Menu button and the title area.

Double-clicking the window Menu button can be used to close the window, unless focus is disallowed in the window. A user must be * able to activate the window Menu button for the window with the * focus by using **KWindowMenu**.

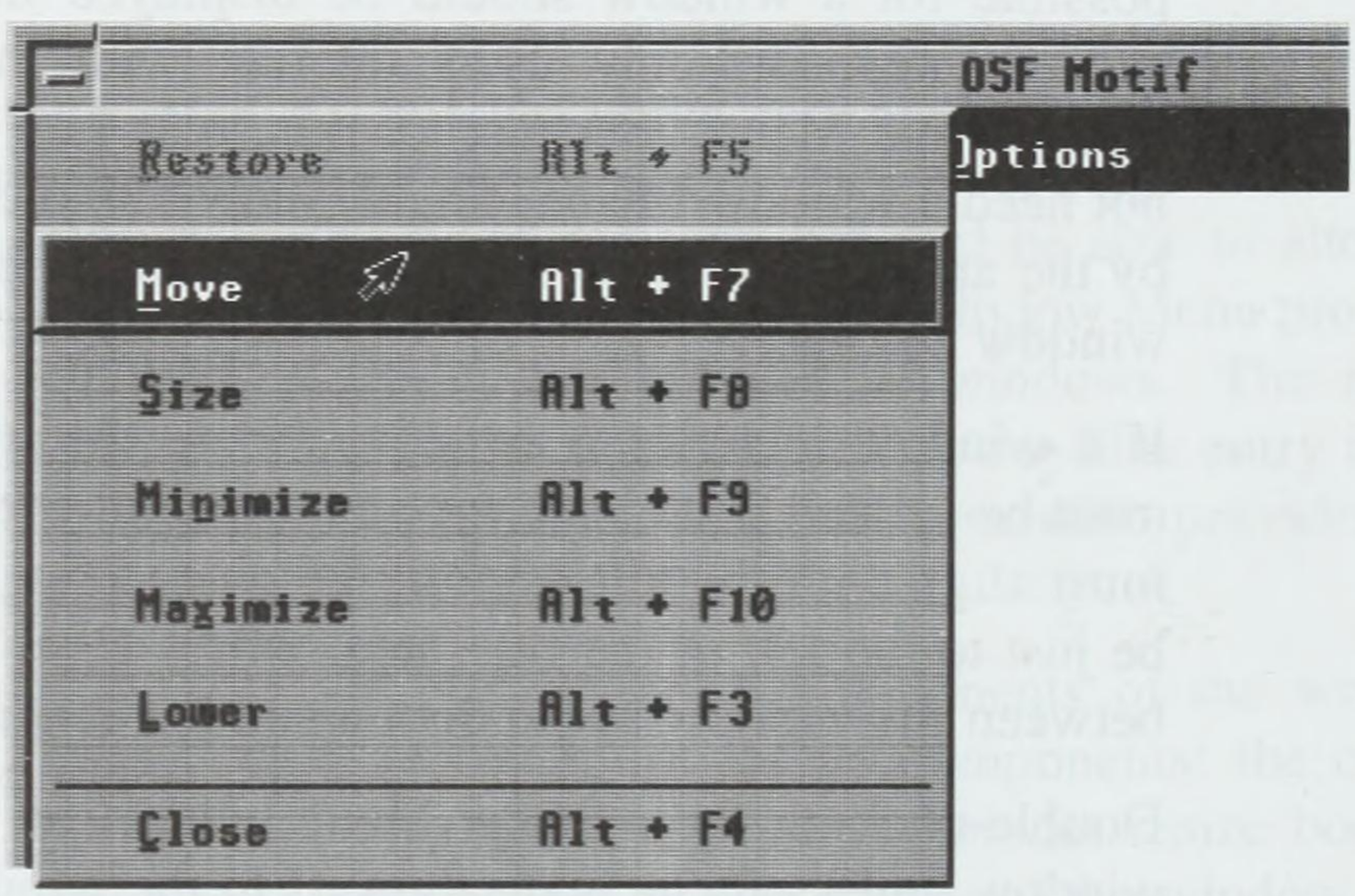
KWindowMenu is usually bound to **MShift KEscape** or **MAlt KSpace** as shown in the following table.

Table 5-1. KWindowMenu Bindings

Virtual Key	Default Binding
KWindowMenu	**MShift <Escape>** **MAlt KSelect**
KSelect	**<Select>** **KSpace** **MCtrl KSpace**
KSpace	**<Space>**

Figure 5-3 shows a typical window Menu.

Figure 5-3. The Window Menu Button with Menu Pulled Down



A primary window Menu must have the following entries in the order listed:

Restore Alt+F5 — Restores a minimized or maximized window to the previous size and location of the window. This entry must be deemphasized (grayed out) when the

	window is in its normal state. This action must have the mnemonic **R**. If this action has an accelerator, it must be **MAlt <F5>** if **<F5>** is available.
Move Alt+F7	Moves a window around the workspace. This action must have the mnemonic **M**. If this action has an accelerator, it must be **MAlt <F7>** if **<F7>** is available.
Size Alt+F8	Changes the height and width of the window in the direction indicated by the pointer. This action must have the mnemonic **S**. If this action has an accelerator, it must be **MAlt <F8>** if **<F8>** is available.
Minimize Alt+F9	Changes a window into an icon. This action must have the mnemonic **N**. If this action has an accelerator, it must be **MAlt <F9>** if **<F9>** is available.
Maximize Alt+F10	Enlarges a window to its maximum size. This action must have the mnemonic **X**. If this action has an accelerator, it must be **MAlt <F10>** if **<F10>** is available.
Lower Alt+F3	Moves a window to the bottom of the window hierarchy. This action can be omitted. This action must have the mnemonic **L**. If this action has an accelerator, it must be **MAlt <F3>** if **<F3>** is available.
Close Alt+F4	Closes a window and removes it from the workspace. This action must have the mnemonic **C**. If this action has an accelerator, it must be **MAlt <F4>** if **<F4>** is available. Applications should prompt the user to save any unsaved changes when a window is closed.

A secondary window Menu must have the following entries in the order listed: **Restore**, **Move**, **Size**, and **Close**. A secondary window Menu can also include **Lower** between **Size** and **Close**, but the

lower option on a secondary window lowers all the windows secondary to that window's primary window. A secondary window must not include an entry for **Minimize**. *

Additional Menu items can be added to the window Menus of both primary and secondary windows according to the guidelines for Menus described in Chapter 4.

5.4 Window Navigation

Moving the focus among windows using the keyboard is called window navigation. Because applications should not warp the mouse pointer, a window manager need not support window navigation when the focus policy is in pointer mode. Window managers must support window navigation when the focus policy is in explicit mode. * * *

The window navigation model can be divided into two levels:

1. Moving among window families (among primary windows)
2. Moving within a window family (among secondary windows)

A window family consists of a single primary window and all of its associated secondary windows. A window manager must support moving the keyboard focus among windows in a window family using the **KNextFamilyWindow** key. **KNextFamilyWindow** is usually bound to **MAlt <F6>** if **<F6>** is available. A window manager may support moving the keyboard focus in the opposite direction from **KNextFamilyWindow** using the **KPrevFamilyWindow** key. **KPrevFamilyWindow** is usually bound to **MShift MAlt <F6>** if **<F6>** is available. * * *

Moving the focus between window families must be done using **KNextWindow** and **KPrevWindow**. **KPrevWindow** should move the focus among the windows in the opposite direction from **KNextWindow**. **KNextWindow** is usually bound to **MAlt KTab**. **KPrevWindow** is usually bound to **MAlt KBackTab**. If there is no icon box, **KNextWindow** and **KPrevWindow** must move among the icons as well as the windows. When focus moves into a window family, the focus should go to the window in the window family that last had the focus. If no window in the family has ever had the * *

focus, it should go to the most recently opened secondary window. The following table shows the common window navigation bindings.

Table 5-2. Window Navigation Bindings

Virtual Key	Default Binding
KNextPane	<F6>
KNextFamilyWindow	MAlt KNextPane
KPrevFamilyWindow	MAlt MShift KNextPane
KNextWindow	MAlt KTab
KPrevWindow	MAlt KBackTab

5.5 Icons

An icon is a stylized representation of an object. A window icon is a minimized representation of a window or window family that can help organize windows and tasks in the display. Iconifying a window is also known as minimizing a window. The window * manager must iconify all windows of window family together. It * must not iconify any single window, primary or secondary, from a * window family without also iconifying all the other windows in the * window family. The iconic representation of a window family should not change any state in the windows, except the visual representation of the window. The application running inside of a * window must continue running even when the application is * iconified. The application can adjust its own state when it is iconified.

5.5.1 Icon Decoration

An icon is made up of an image and a label. Figure 5-4 shows a typical Motif icon.

Figure 5-4. A Typical OSF/Motif Icon

The image must be surrounded by a border that indicates when the icon has the keyboard focus. The image area can contain text or a bitmap. The label must be located just below the image and its border. The label can also indicate when the icon has the keyboard focus by highlighting along with the image area.

The label should contain the same text as the title area of the corresponding primary window, or an abbreviated form of it. When the icon does not have the keyboard focus, the width of the label must be the same as the width of the image area and its border, truncating text if necessary. When the icon has the keyboard focus, the width of the label can expand to display the entire text.

5.5.2 Icon Menu

Clicking **BSelect** in an icon must give the icon the keyboard focus and post the icon Menu. The icon Menu must be the same Menu as the window Menu for the associated primary window. **Size** and **Minimize** must not be available from the icon Menu. Navigating to the icon must also give the icon the keyboard focus and post the icon Menu.

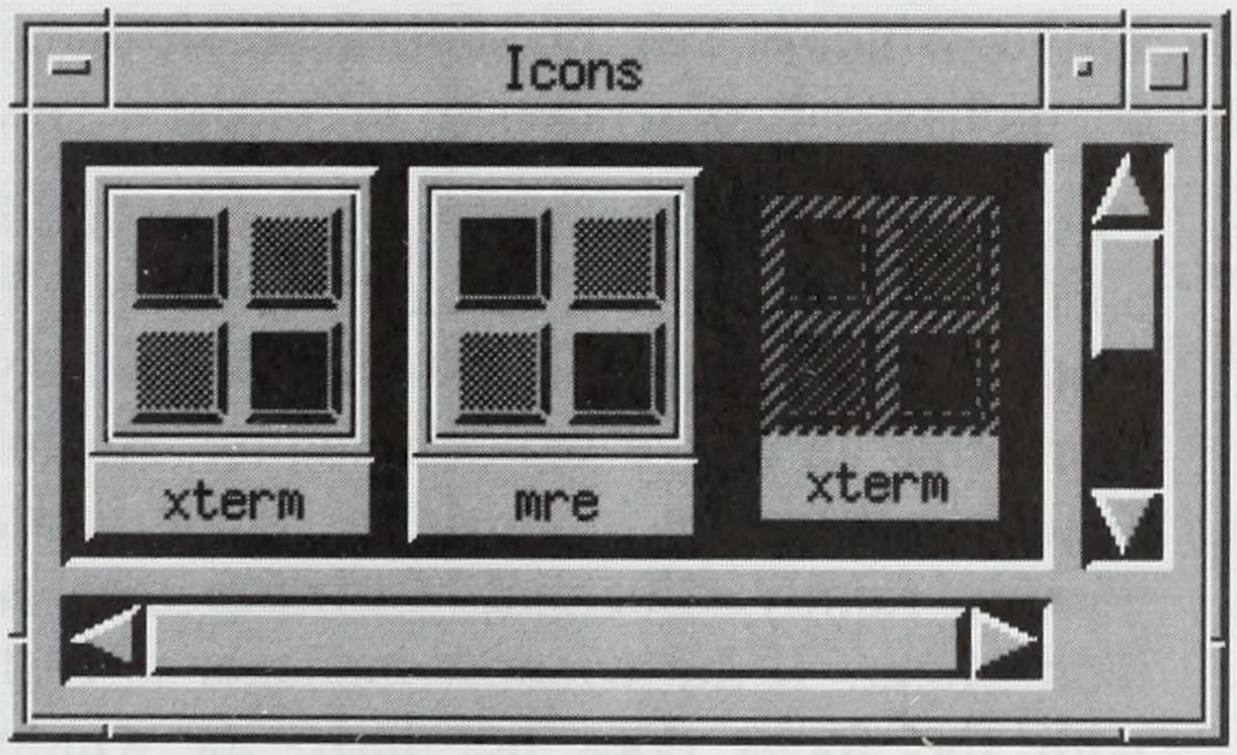
Double-clicking **BSelect** anywhere in the icon must restore the window family, just as the icon Menu item **Restore** does. Pressing **BSelect** or **BDrag** anywhere in the icon and dragging the mouse pointer must move the icon to track the pointer, just as the icon Menu item **Move** does.

If the Window Manager has been customized by the user so that the icon Menu does not popup when the icon gets the focus, **KWindowMenu**, **KMenu**, and **BMenu** must popup the icon Menu. *

5.5.3 Icon Box

An icon box is a specialized window that acts as a storage location for icons. An icon box acts like a typical window in the sense that it has a window frame and frame components. The client area of an icon box must have an area for holding icons and horizontal and vertical scroll bars for moving around the icon area. Figure 5-5 shows a typical icon box. *

Figure 5-5. A Typical OSF/Motif Icon Box



The icon box must have all the same components as any other primary window. Like other windows it can be sized, moved, minimized, maximized, restored, and lowered. However, the window manager must not allow the icon box to be closed. The system Menu action **Close** must be replaced with the system Menu action **Pack Icons** in an icon box. Double-clicking **BSelect** in the icon box system Menu must only open the icon box system Menu. It must not close the icon box. **Pack Icons** arranges the icons as close as possible together in the visible icon area if possible. **Pack Icons** must have the mnemonic **P**. If **Pack Icons** has an accelerator, it must be **MAlt <F12>** if **<F12>** is available. When the input focus is in the icon box, **KWindowMenu** must pop up the icon box system Menu if the Menu is unposted. If the Menu is already posted, **KWindowMenu** must unpost the Menu.

The icon box can contain an icon for each window family, even if it is active. The icon for minimized window families must be the same as the icon would be outside of the icon box. The icon for an active window family should be similar to the minimized icon, deemphasized somehow.

Clicking **BSelect** in an icon must give the icon keyboard focus and post the icon Menu. The icon Menu must be the same Menu as the window Menu for the associated primary window. If the icon represents an active window family, **Restore** and **Size** must not be available from the icon Menu. If the icon represents a minimized window family, **Size** and **Minimize** must not be available from the icon Menu. If the window manager is customized so that the icon with the focus does not have its window Menu automatically posted, **BMenu** or **KMenu** must post its Menu.

Double-clicking **BSelect** anywhere in an icon that represents a minimized window family must restore the window family, just as the icon Menu item **Restore** does. Double-clicking **BSelect** anywhere in an icon that represents an active window family must raise the window family to the top of the window hierarchy. Pressing **BSelect** or **BDrag** anywhere in the icon and dragging the mouse pointer must move the icon within the icon area to track the pointer, just as the icon Menu item **Move** does.

KDown, **KUp**, **KRight** and **KLeft** must navigate among the icons in the icon box. **KDown**, **KUp**, **KRight** and **KLeft** are usually bound to the arrow keys, as shown in the table at the end of this section. Icon navigation must behave as described in the following text. Note that **KDown** and **KRight** do not need to traverse icons in the same order. *

KDown In a left-to-right direction language environment, must move the location cursor through the icons in the icon box, starting at the upper-left icon and ending at the lower-right icon then wrapping back up to the upper-left. In a right-to-left direction language environment, the location cursor can move, starting at the upper-right and moving to the lower-left. *

KLeft Must move the location cursor through the icons in the opposite direction of **KRight**. *

KRight In a left-to-right direction language environment, must move the location cursor through the icons in the icon box, starting at the upper-left icon and ending at the lower-right icon then wrapping back up to the upper-left. In a right-to-left direction language environment, the location cursor can move, starting at the upper-right and moving to the lower-left. *

KUp Must move the location cursor through the icons in the opposite direction of **KDown**. *

Traversal can be either through all the icons in the icon box, regardless of whether they are in the visible portion of the icon box, or only through the icons in the visible portion of the icon box. If traversal only goes through the visible portion of the icon box, the ScrollBars in the icon box must be traversable using **KNextField** and **KPrevField**, which move the location cursor among the ScrollBars and the icon area, if the icon area contains any visible icons. If traversal goes through all icons in the icon box, the ScrollBars should not be traversable, and the visible portion of the icon area must be scaled to keep the location cursor visible. *

Table 5-3. Icon Box Traversal Bindings

Virtual Key	Default Bindings
KTab	<Tab>
KBackTab	MShift <Tab>
KNextField	KTab
	MCtrl KTab
KPrevField	KBackTab
	MCtrl KBackTab
KWindowMenu	MShift KEscape
	MAlt KSpace
KMenu	<Menu> or <F4>
KDown	<↓>
KLeft	<←>
KRight	<→>
KUp	<↑>

Chapter 6

Designing for International Markets

This chapter provides basic guidelines for producing applications for international markets. Internationalization is the process of generalizing programs or systems so that they can handle a variety of languages, character sets, and national customs. Localization is the process of providing language- or country-specific information or support for programs.

In general, internationalization issues are handled by tools available to programmers on their system. For example, ANSI C standard (ANS X3.159-1989) and POSIX 1003.1 have defined internationalization in terms of locale. The locale can then be set as part of the user's environment, allowing the program to access locale-specific information, such as data formats, collating sequences, and system messages, from system- or application-specific databases. You should use any internationalization tools available on your system to support internationalization in your application.

Following are some of the issues that are handled by an internationalized application. Note that your application generally does not need to directly deal with these issues. In most cases, they should be addressed by the internationalization tools available on your system. They are provided here primarily to increase your

awareness of issues that can affect your programming. In a few cases, you may need to adjust your program to allow for size and layout changes of data in different locales.

- Collating sequences
- Country-specific data formats
- Icons, symbols, and pointer shapes
- Scanning direction
- Modularized software
- Translation of screen text

6.1 Collating Sequences

To produce an alphanumeric list, printable characters are sorted according to a collating sequence. Printable characters include letters possibly with accents, numbers, punctuation characters, and other symbols such as an * (asterisk) or & (ampersand). The collating sequence defines the value and position of a character relative to the other characters.

Many applications make frequent use of collating sequences to produce alphanumeric lists. Examples of alphanumeric lists include the following:

- A directory listing of filenames
- The output from a sorting utility
- An index produced by a text-processing application
- The lists produced by a database application, such as lists of names or addresses

6.2 Country-Specific Data Formats

Country-specific data formats include the following:

- Thousands separators
- Decimal separators (or, radix characters)
- Grouping separators
- Positive and negative values
- Currency
- Date formats
- Time formats
- Time zones
- Telephone numbers
- Proper names and addresses

6.2.1 Thousands Separators

The comma, period, space, and apostrophe are examples of valid separators for units of thousands as shown in the following examples:

1234567
1 234 567
1.234.567
1'234'567
1,234,567

6.2.2 Decimal Separators

The period, comma, and the center dot are examples of valid separators for decimal fractions as shown in the following examples:

5
5,324
5.324
5 324
5·324

6.2.3 Grouping Separators

Grouping may not be restricted to thousands separators as shown in the following examples:

400,001.00
40,0001,00

6.2.4 Positive and Negative Values

Various countries indicate positive and negative values differently. The symbols + and - may appear either before or after the number. Negative numbers may be enclosed in parentheses in applications such as a spreadsheet.

6.2.5 Currency

Currency formats differ among various countries. The comma, period, and colon are examples of valid separators for currency. There may be one or no space between the currency symbol and the amount. The currency symbol may be up to four characters. The following example shows valid currency values:

Sch3.50
SFr. 5.-
3.50FIM
25 c
3F50
760 Ptas
Esc. 3.50
kr. 3,50

6.2.6 Date Formats

Most countries use the Gregorian calendar, but some do not. Dates can be formatted differently based on the locale. Separators can be different in different locales or left out altogether. The hyphen, comma, period, space, and slash are all examples of valid separators for the day, month, and year. In numeric date formats the month and day fields can be reversed, and in some cases, the year field can come first. For example, the 4th of August 1990 can be written as either 4/8/90 or 8/4/90 depending on locale. In addition, users in Sweden sometimes place the year first, so June 11, 1990 could be 900611 or 901106.

6.2.7 Time Formats

Time formats can change based on locale. The colon, period, and space are examples of valid separators for hours, minutes, and seconds. The letter **h** can separate hours and minutes. There is both 12- or 24-hour notation. For 12-hour notation, a.m. or p.m. can appear after the time, separated by a space. The following example shows a number of valid time formats:

1830
18:30
04 56
08h15
11.45 a.m.
11.45 p.m.
13:07:31.30
13:07:31

6.2.8 Telephone Numbers

Telephone numbers can contain blanks, commas, hyphens, periods, and square brackets as valid separators, for example. Telephone numbers can be displayed in local, national, and international formats. Local formats vary widely. National formats may have an area code in parentheses, while the international format may drop the parentheses, but add a plus sign at the beginning of the number to indicate the country code. The following examples show valid telephone number formats:

(038) 473589
+44 (038) 473549
617.555.2199
(617) 555-2199
1 (617) 555-5525
(1) 617 555 5525
911
1-800-ORDERME

6.2.9 Proper Names and Addresses

Addresses may vary from two to six lines long, and can include any character used in the locale's character set. The post code (zip code) can be in various positions in the address, and can include alphabetic characters and separators as well as numbers.

6.3 Icons, Symbols, and Pointer Shapes

It may not always be possible to design an icon, pointer shape, or other graphical symbol that adequately represents the same object or function in different countries. Culture is inherent even in seemingly universal symbols. For example, sending and receiving mail is a commonly understood function, but representing that function with an icon of a mail box may be inappropriate because the appearance of mail boxes varies widely among countries. Therefore, an envelope may be a more appropriate icon. You should make sure that graphical symbols are localizable.

When used correctly, graphical symbols offer the following advantages:

- They are language independent and do not need to be translated. In some cases, you may not be able to avoid changing an icon or symbol for a culture that is vastly different. However, design icons and symbols with the entire user population in mind so that you can try to avoid redesigning.
- They can be used instead of computer terms that have no national-language equivalent.
- They may have more impact when used with text as warnings than the text alone.

Here are a few guidelines to follow when creating icons, symbols, or pointer shapes:

- Use an already existing international icon, if possible.
- Make your icons, symbols, or pointer shapes represent basic, concrete concepts. The more abstract the icon, the more explanatory documentation is needed.
- Check your icons and symbols for conflicts with existing icons or symbols for that function.
- Do not incorporate text in icons because the text will need to be translated. Translated text expands, and might no longer fit the icon.
- Test and retest your symbols and icons in context with real users.

6.4 Scanning Direction

Readers of Western languages scan from left to right across the page (or display screen) and from top to bottom. In other languages, particularly Eastern ones such as Hebrew and Arabic, this is not the case; readers scan from right to left. The scanning direction of the country of localization may have an impact on the location of components in dialog boxes, the order of selections in menus, and other areas.

If your application will be used in environments other than those that scan from left to right, remember that the the scanning direction should match the input direction.

6.5 Designing Modularized Software

Modularizing software allows for easier localization; that is, a properly modularized application requires that fewer files be modified to localize the application. Guidelines for designing modularized software are as follows:

- Create separate modules for text, code, and input/output components that need to be changed to accommodate different markets.
- Separate all user interface text from the code that presents it.
- Use standard (registered) data formats, such as ISO and IEEE.
- Use standard processing algorithms for all processing, storage, and interchange.

In general, you should modularize your application so that elements that need to be translated to different languages are in separate files, and that those files are the only files that will need changes for localization. Furthermore, you should have a different set of language-dependent text files for each locale that are read in at run time using the internationalization tools available on your system. In this way you can support multiple locales simultaneously.

6.6 Translating Screen Text

Well-written screen text makes an application easier for users to understand. It also makes translation easier.

Use the following guidelines to write screen text for translation:

- Write brief and simple sentences; they are easy to understand and translate.
- Write affirmative statements; they are easier to understand than negative statements. For example, use "Would you like to continue?" rather than "Wouldn't you like to continue?"

- Use active voice; it is easier for both application users and application translators to understand. For example, use "Press the Help button." rather than "The Help button should be pressed."
- Use prepositions to clarify the relationship of nouns; avoid stringing three or more nouns together.
- Use simple vocabulary; avoid using jargon unless it is a part of your audience's working vocabulary.
- Allow space for text expansion. Text translated from English is likely to expand 30% to 50%, or even more.

Chapter 7

Controls, Groups, and Models Reference Pages

This chapter presents detailed information about components, user interface models, and concepts in reference format. Each topic starts on a new page and is organized alphabetically. Details on user interface models and concepts are provided in the earlier chapters of this *Style Guide*. The model and concepts reference pages are provided here only as a quick reference to information provided earlier, and are not complete in every detail. Therefore, they should not be used as the definitive source for information about user interface models and concepts. This chapter includes the following model and concepts reference pages.

Accelerators	File Menu	Layout Groups
Activation	Focus	Mnemonics
Basic Controls	Framing Groups	Navigation
Basic Groups	Help Menu	Pointer Shapes
Default Activation	Icon Menu	Primary Selection
Drag and Drop	IconBox	Quick Transfer
Edit Menu	Icons	Selection
Field Controls	Input Devices	Window Menu

The reference pages for components are provided for designers to use when implementing the components described in this guide or when creating new components. Remember when designing new

components, you should follow the same rules that application designers follow. As such, you should be familiar with all the chapters of this *Style Guide*, not just this reference-page section. New components must follow the guidelines for designing applications. This chapter contains the following control and group reference pages.

Canvas	Label	RadioButton
CascadeButton	List	Sash
CheckButton	MainWindow	Scale
CommandBox	MenuBar	ScrollBar
CommandDialog	Menus	ScrolledWindow
Composition	MessageDialog	SelectionBox
DialogBox	OptionButton	SelectionDialog
ErrorDialog	PanedWindow	Separator
FileSelectionBox	Panel	Text
FileSelectionDialog	PromptDialog	ToggleButton
Frame	PushButton	WarningDialog
InformationDialog	QuestionDialog	WorkingDialogBox

When designing a new component, compare its features with those of other components. If the new component has a feature that is the same as another component, the mechanisms for using the feature, layout, key bindings, graphics, and so on, must be similar to the existing component.

Each component reference page contains the following information about the component.

Description A description of the use and appearance of each component, group, or model.

Illustration An illustration of a typical component. The illustrations in this *Style Guide* use the OSF/Motif reference appearance with 3-dimensional beveled edges. Although it is important to be consistent in the placement of the elements in a component, the appearance, or rendering, of the component is not an issue of *Style Guide* compliance.

Navigation A description of the methods for navigating within a component.

Other Operations A description of other operations available within the component.

Common Bindings A list of the virtual keys used by this component, and the common substitutions for each virtual key.

Related Information A list of sections and related reference pages in this *Style Guide* with additional information.

Accelerators

Description

An accelerator is a key or key combination that invokes the action of some component without the location cursor being on the component when the accelerator is pressed. Accelerators are most commonly used to activate Menu items without first posting the Menu. You should provide accelerators primarily as a matter of utility, not for design conformity.

If the button with an accelerator is within a primary or secondary window, or within a PullDown Menu system from its MenuBar, it must be activatable whenever the input focus is in the window. If the button with an accelerator is within a Popup Menu system, it must be activatable whenever the focus is in the component with the Popup Menu. *

Applications can provide accelerators for any button component. Implementations must support accelerators in PushButtons and ToggleButtons that are in Menus. If a button has an accelerator, the accelerator must be shown following the label of the button. *

Activation

Description

This reference page only provides a short description of the various types of activation. Chapter 3 describes each type in detail.

Basic Activation

All activatable basic controls must be activated using basic activation. Clicking **BSelect** on the button must activate the button. Pressing **KActivate** and **KSelect** on the button with the location cursor must activate the button. **KActivate** has a different function when a default button is available.

Accelerators

An accelerator is a key or key combination that invokes the action of some component without the location cursor on the component when the accelerator is pressed. Accelerators are most commonly used to activate Menu items without first posting the Menu.

Mnemonics

A mnemonic is a single character that can be associated with any component that contains a text label. When the location cursor is on a component within a Menu, a MenuBar or the same field as a component with a mnemonic, typing the mnemonic character must move the location cursor to the component and activate it. If a mnemonic is used for an OptionButton, for a CascadeButton in a MenuBar, or a PushButton that is not in a basic group (that is, not in a Panel, a Menu, or a MenuBar), typing **MAlt** and the mnemonic anywhere in the window or its Menus must move the cursor to the component with that mnemonic and must activate it.

Help Activation

Pressing **KHelp** on a component must invoke any context-sensitive help for the component or its nearest ancestor with context-sensitive help available. Within DialogBoxes, context-sensitive help should provide help information or the DialogBox as a whole. *

Popup Menu Activation

Clicking **BMenu** in a component with a Popup Menu must post the Popup Menu. Pressing **BMenu** in a component with a Popup Menu must post the Popup Menu in a spring-loaded manner until the button is released or a selection is made. *

Default Activation

In a DialogBox, pressing **KActivate** anywhere except in a PushButton component or double-clicking **BSelect** in an activatable field component, such as Text or List, must activate the default PushButton in the DialogBox. *

Expert Activation

Some elements, usually PushButtons and ToggleButtons, can have expert actions associated with them. **BSelect Click 2** must activate any expert action for the element. Expert action should only be available in a Panel, and the expert action of all the buttons should be similar. The expert action should include the regular action of the component in a more global manner. *

Common Bindings

Virtual Key	Default Bindings
KEnter	<Return> <Enter>
KActivate	KEnter MCtrl KEnter
KSelect	<Select> <Space> MCtrl <Space>
KHelp	<F1> <Help>

Related Information

See Chapter 3 for more information about the activation model.

Basic Controls

Description

Basic controls are components that only take simple input. They are distinct from field controls in that they are usually elements of navigation fields rather than fields themselves. Basic controls must have no internal navigation. The following text describes the basic controls. *

Separator — Draws a separating line within windows, between Menu items, and between Panes of a PanedWindow. A Separator allows no application interaction.

Label — Displays static text and images. A Label presents application information to users.

PushButton — A button used to activate an operation. A PushButton contains a label that indicates the operation of the button. The label can contain text or an image.

CascadeButton — A button used to display a PullDown Menu. A CascadeButton contains a label that indicates the Menu displayed. CascadeButtons can also contain an arrow graphic after the label to distinguish it from PushButtons and to indicate the direction of the cascading Menu.

OptionButton — A button used to display an Option Menu. An Option Menu allows for a one of many selection. An OptionButton contains a label that indicates the current state of the Option Menu, and a bar graphic to distinguish it from a PushButton.

ToggleButton — A button with two states: on and off. A ToggleButton contains a label that indicates the state of the ToggleButton when it is set.

Normally, preceding the label is a graphic indicator of the state of the ToggleButton.

CheckButton A ToggleButton in a group of ToggleButtons where any number of the ToggleButtons can be on at a time. The graphic indicator for a CheckButton is usually a raised square to indicate the on state or a pressed square to indicate the off state.

RadioButton A ToggleButton in a group of ToggleButtons where only one of the ToggleButtons can be on at a time. The graphic indicator for a RadioButton is usually a filled diamond or circle to indicate the on state or an empty diamond or circle to indicate the off state.

A single line Text control can be configured to act like a basic control.

Navigation

Basic controls must have no internal navigation. *

Other Operations

Activatable basic controls follow the basic activation model described in Section 3.3.1.

Related Information

For more information, see the reference pages for each basic control.

Basic Groups

Description

Basic groups of controls are used to organize groups of basic controls. The following text describes the basic groups.

Panel Organizes a collection of basic controls in a horizontal, vertical, or 2-dimensional layout. A Panel is usually composed of just one type of control.

Menu Organizes a collection of buttons, labels, and separators in a horizontal, vertical, or 2-dimensional layout within a separate Menu window. There are three types of Menus: PullDown, Popup, and Option. A Menu is only available while it is popped up or pulled down.

MenuBar Organizes a collection of CascadeButtons in a horizontal layout at the top of a MainWindow.

Navigation

KDown, **KLeft**, **KRight**, and **KUp** must navigate within a basic group according to the navigation model described in Chapter 2. *

Common Bindings

Virtual Keys	Default Bindings
KDown	<↓>
KLeft	<←>
KRight	<→>
KUp	<↑>

Related Information

For more information, see the reference pages for each basic group.

Canvas

Description

A Canvas should be used to present and edit graphics. This component must be composed of an area for presenting and editing graphics. *

Navigation

Canvas navigation is unspecified.

Other Operations

Most Canvas operations are unspecified. A description of the specified Canvas operations follows:

KHelp Must provide any available help for the Canvas. *

Common Bindings

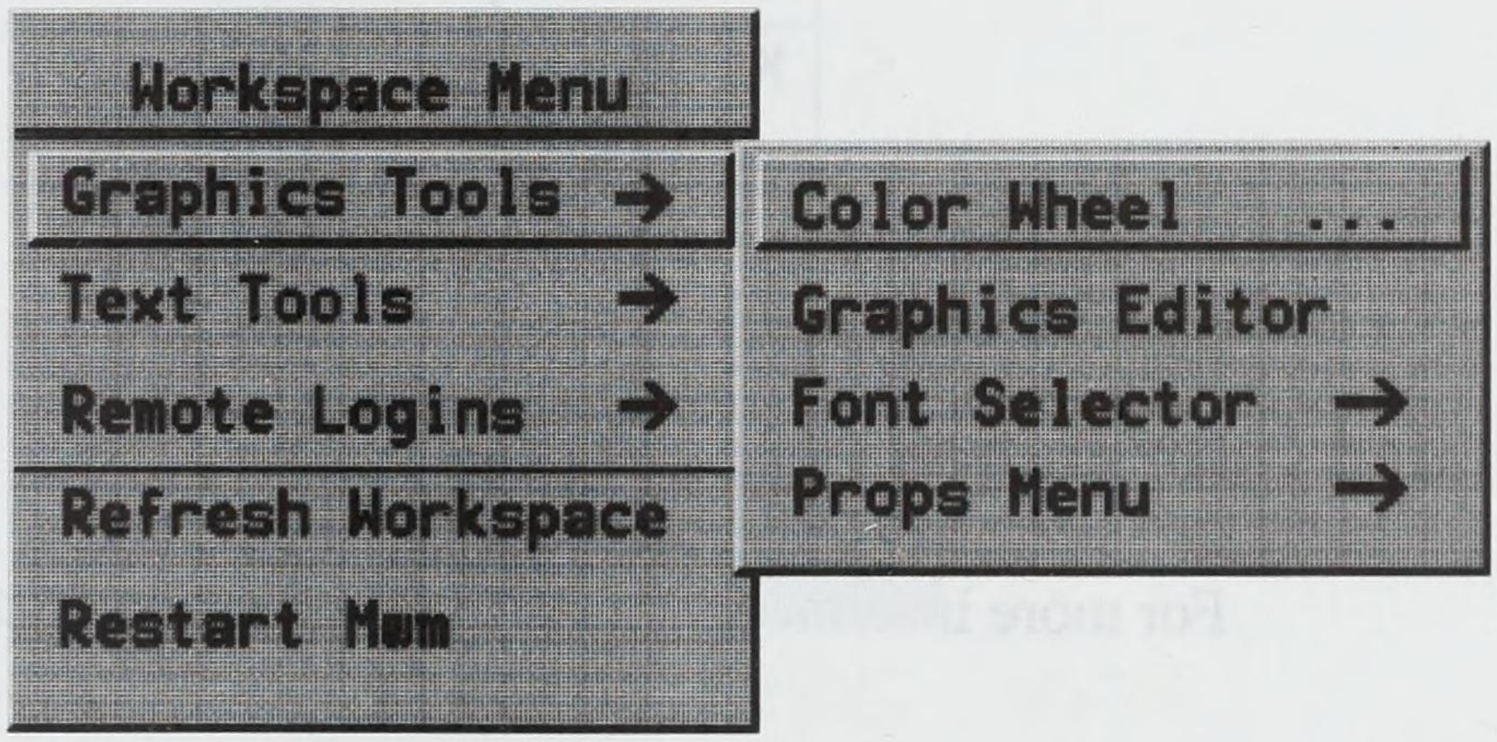
Virtual Keys	Default Bindings
BSelect	<Btn1>
KHelp	<Help> <F1>

CascadeButton

Description

A CascadeButton should be used to post a PullDown Menu. This component must be composed of a button, with either a text or graphics label. Following the label, this component should also include an arrow graphic, pointing in the direction that the Menu will be posted to distinguish it from a PushButton. The graphic is usually not shown in a MenuBar.

Illustration



Navigation

A CascadeButton must have no internal navigation. *

Other Operations

CascadeButtons follow the Menu activation model described in Section 2.3.3. The following text describes the other operations of this component.

KHelp Must provide any available help for the CascadeButton. *

Common Bindings

Virtual Keys	Default Bindings
KHelp	<Help> <F1>

Related Information

For more information, see the reference pages for Menus.

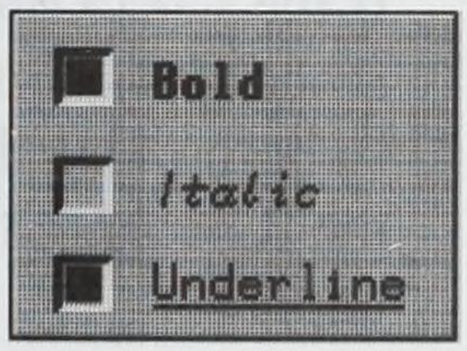
CheckButton

Description

A CheckButton should be used to set options in the application. A CheckButton is a special case of a ToggleButton. Any number of CheckButtons can be set at the same time.

This component must be composed of a text or graphic label, and a graphic that indicates the state of the CheckButton. The graphic should be a square preceding the label, and should have two distinctive states indicating the set and unset states of the CheckButton. On color systems, the on state color can be distinct from general application colors to visually distinguish the on state. *

Illustration



Navigation

CheckButtons must have no internal navigation. *

Other Operations

The following text describes the CheckButton operations.

BSelect Press

Must arm the CheckButton. If the CheckButton was previously unset, it must show the CheckButton in the set state. If the CheckButton was previously set, it must show the CheckButton in the unset state.

BSelect Release

If the release happens in the same CheckButton that the press occurred in, then if the CheckButton was previously unset, it must be set, or if the CheckButton was previously set, it must be unset. In all cases the CheckButton must be disarmed, and if the CheckButton was in a Menu, the Menu must be unposted.

KActivate If the CheckButton was previously unset, it must be set. If the CheckButton was previously set, it must be unset. In both cases the CheckButton must be disarmed, and if the CheckButton was in a Menu, the Menu must be unposted.

KSelect If the CheckButton was previously unset, it must be set. If the CheckButton was previously set, it must be unset. In both cases the CheckButton must be disarmed, and if the CheckButton was in a Menu, the Menu must be unposted.

KHelp Must provide any available help for the CheckButton.

Common Bindings

Virtual Keys	Default Bindings
BSelect	<Btn1>
KEnter	<Return>
KActivate	<Enter>
	MCtrl <Enter>
KSelect	<Select>
	<Space>
	MCtrl <Space>
KHelp	<Help>
	<F1>

Related Information

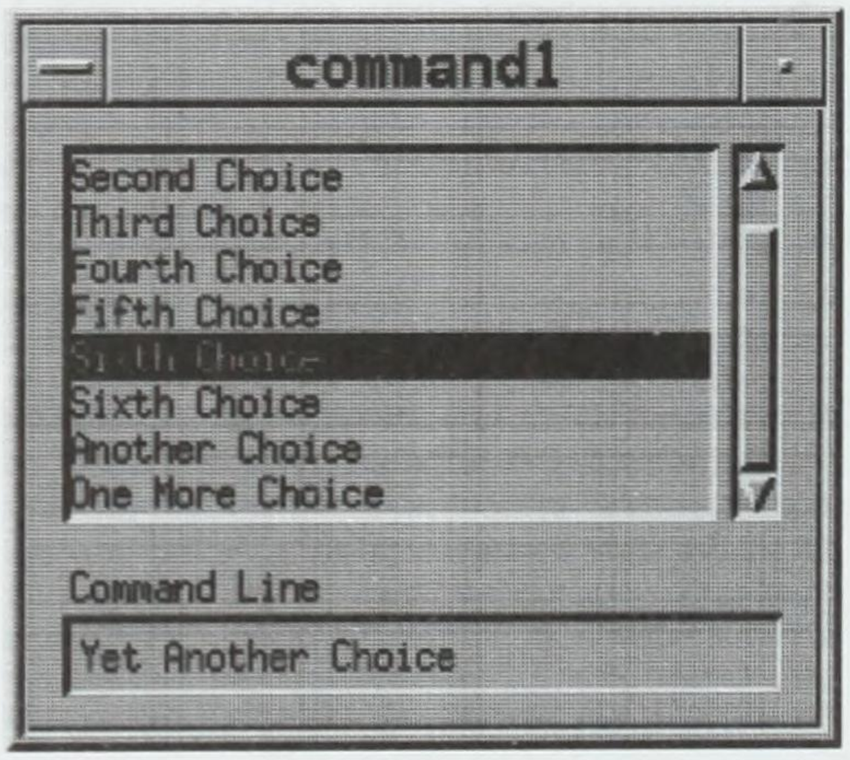
For more information, see the reference pages for RadioButton and ToggleButton.

CommandBox

Description

A CommandBox is a special purpose composite component for command entry that provides a built-in command history mechanism. The CommandBox must be composed of a Text component with a command line prompt for command input, and a List component above the Text component for a command history area. When a command is entered in the Text component, it must be added to the end of the List component and made visible. This must not change the state of the selection in the List if one exists. The List component can be scrollable.

Illustration



Related Information

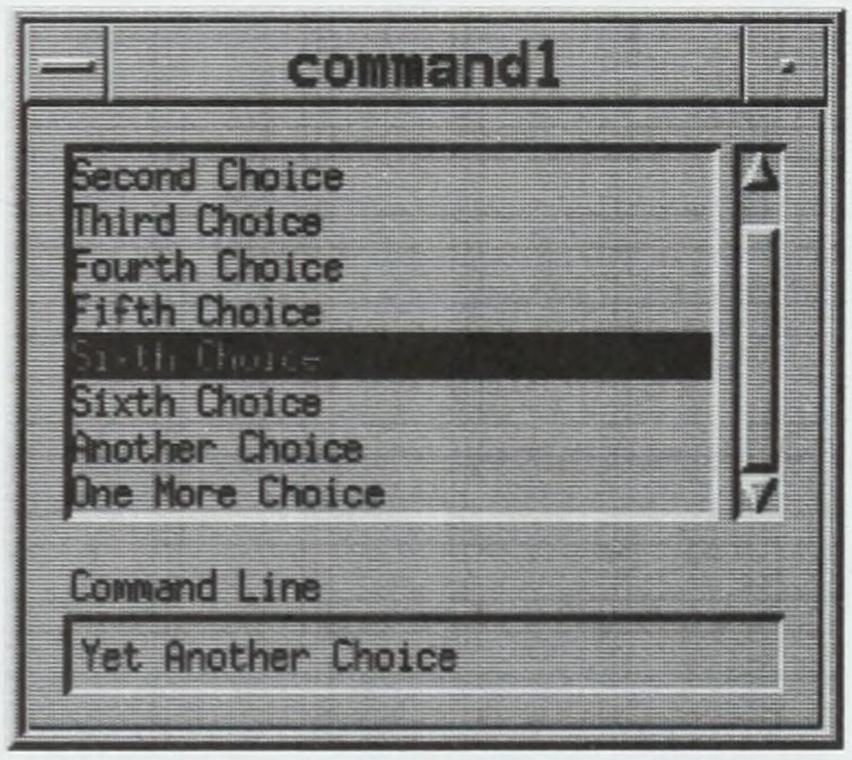
For more information, see the reference page for Command DialogBox.

CommandDialog

Description

A FileSelectionDialog should be used to enter commands for processing. It should not interrupt the user's interaction with the application. It should include a CommandBox.

Illustration



Related Information

For more information, see the reference page for DialogBox.

Composition

Description

The Composition group should be used to organize components in an arbitrary layout. The layout style can be either arbitrary, in even rows and columns, or so that the position of components is relative to the Composition component and the components it contains. This component is composed of an area for organizing components.

Navigation

This group must follow the navigation model as described in * Chapter 2.

Related Information

For more information, see the reference pages for each layout group.

Default Activation

Description

The default action of a DialogBox is activated according to the following rules:

- When the input focus is on a PushButton other than the default, and if the component accepts input, pressing **KActivate** must activate the PushButton. The highlighting must be removed from the default PushButton. The highlighting can be moved to the new PushButton. When the focus policy is implicit, the changes in highlighting should have a delay to prevent flicker when moving the pointer through the components. *
- When the focus is on an activatable field control, such as List or Text, and if the control accepts input, pressing **BSelect Click 2** or **KActivate** must activate both the field control's corresponding action and the default action of the DialogBox. *
- When the input focus is on any other component, and if the component accepts input, pressing **KActivate** must activate the default action of the DialogBox, but not the action of the component. *
- When the focus is not on any component, pressing **KActivate** must activate the default action of the DialogBox. *

Related Information

See Chapter 3 for general information about the activation model and default activation.

DialogBox

Description

A DialogBox should be used to group components in a window secondary to the main tasks of the application. Although a DialogBox may contain any components, a simple DialogBox is composed of a Label and PushButtons for supplying a response to the DialogBox. The OSF/Motif toolkit provides a number of ready-designed DialogBoxes for common uses: CommandDialog, FileSelectionDialog, MessageDialog, PromptDialog, and SelectionDialog. There are also a number of different types of MessageDialogs: ErrorDialog, InformationDialog, QuestionDialog, WorkingDialog, and WarningDialog. The illustration on this reference page shows a typical DialogBox.

DialogBox PushButtons should use the following common labels and actions. The actions are in the approximate sequence in which they should appear in DialogBoxes.

Yes Must indicate an affirmative response to a question posed in the DialogBox, and then close the window. While **Yes** is not an active-voice label, it implies a positive response to a question in a **QuestionDialog**. Only use **Yes** if it is a clear answer to the question. *

No Must indicate a negative response to a question posed in the DialogBox, and then close the window. While **No** is not an active-voice label, it implies a positive response to a question in a **QuestionDialog**. Only use **No** if it is a clear answer to the question. *

OK Must cause the application to apply any changes made in the DialogBox and dismiss the DialogBox. While **OK** is not an active-voice label, its usage is too common to change. *

Apply Must apply any changes made to components in the DialogBox. *

Retry Must cause the task in progress to be attempted again. * This action is commonly found in message boxes that report an error.

Stop Must end the task in progress at the next possible * breaking point. This action is commonly found in progress message boxes.

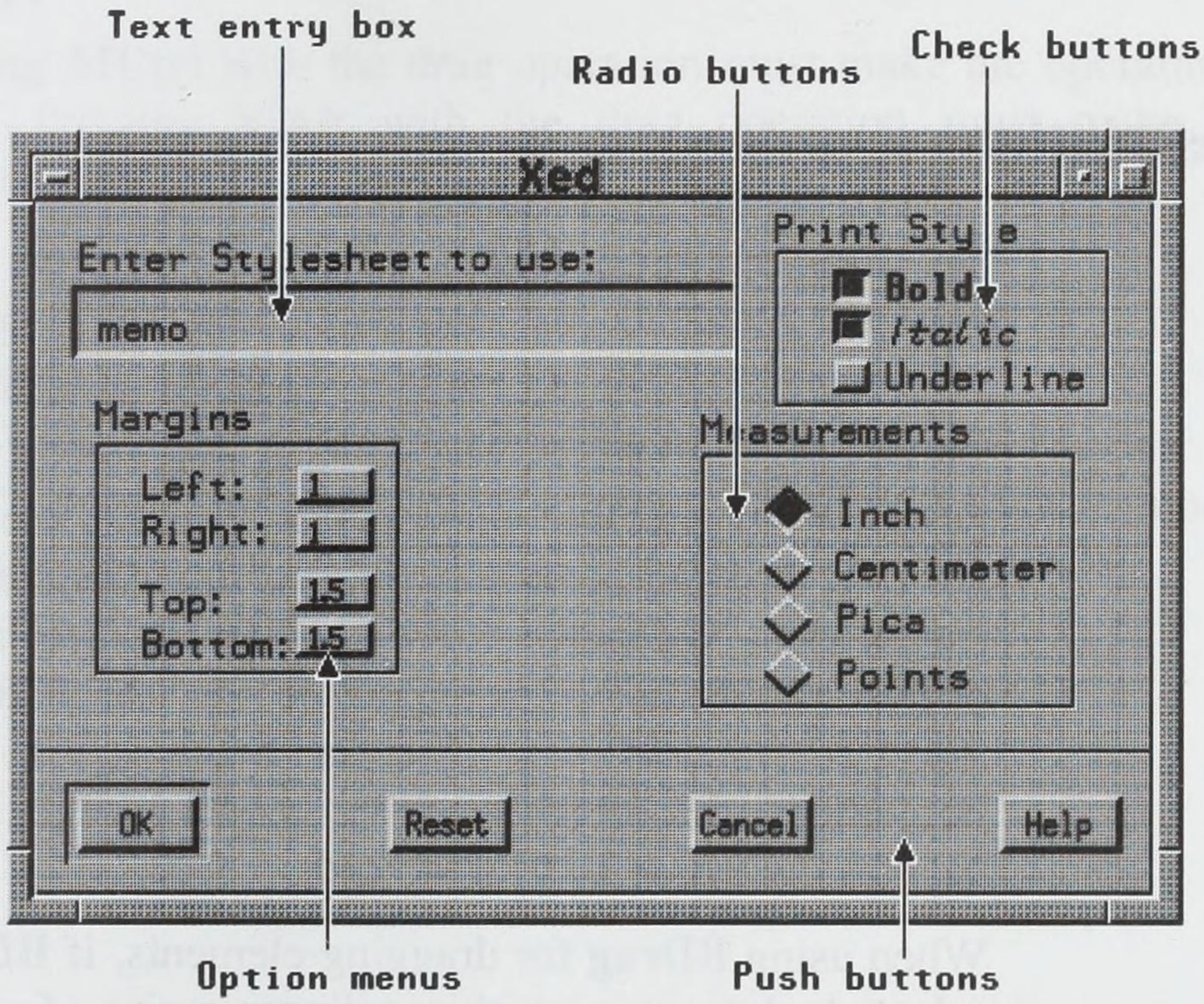
Reset Must cancel any user changes that have not been applied * to the application. It must also reset the status of the * DialogBox to the state since the last time the DialgoBox * action was applied or to the initial state of the DialogBox.

Cancel Must close the DialogBox without performing any * DialogBox actions not yet applied to the application. * Pressing **KCancel** anywhere in the DialogBox, except * during a cancelable drag operation, must perform the * action of this button.

Help . . . Must provide any help for the DialogBox. The ellipses * should not be used if the help action is not provided in a new DialogBox.

You should arrange PushButton actions in DialogBoxes like other PushButtons, according to order and frequency of use. The common action PushButtons should be ordered as presented in the previous section. Positive responses to the DialogBox should be presented first, followed by negative responses and canceling responses. **Help** should always be the last action on the right.

Illustration



Navigation

A DialogBox must follow the navigation model described in * Chapter 2.

Related Information

For more information, see the reference pages for CommandDialog, ErrorDialog, FileSelectionDialog, InformationDialog, MessageDialog, PromptDialog, QuestionDialog, SelectionDialog, WarningDialog, and WorkingDialog.

Drag and Drop

Description

The drag-and-drop model allows the user to move or copy the selected elements by dragging them to a new location. Even collections that do not support dragging of their own elements can support drops of elements from other collections.

If a component implements dragging, it should use **BDrag** to perform the drag operations. Some implementations may choose to use **BSelect** instead of **BDrag**. When **BSelect** is used for dragging * operations, its use must supersede the use of **BSelect Drag** in the * selection models described in Section 3.1. Applications that * implement the drag-and-drop model using **BSelect** must also * support the use of **BDrag** instead.

When using **BDrag** for dragging elements, if **BDrag** is pressed on a * selected element or within a linear range of selected elements, it * must drag the entire selection until it is released. If **BDrag** is * pressed outside of the selected range in a text-like collection, it * must do a quick transfer instead of dragging the element. Quick transfer is described in Section 3.2.3. If **BDrag** is pressed on an unselected element in a list-like collection, it can either do a quick transfer or a dragging operation. The behavior of **BDrag Press** in a graphics-like collection is unspecified. It can either do a quick transfer, a dragging operation, or pan the visible region.

The type of the elements dragged, and the source and destination of the drag-and-drop operations determine whether the operation causes a move or a copy as follows:

- Dragging to a transport device, such as a printer or mailbox, * must be treated as a copy.
- Dragging to a removal device, such as a wastebasket or * shredder, must be treated as a move.

- Dragging to a non-Text container must be treated as a move. *
- Dragging text to or from a Text component must be treated as a copy. *

Pressing **MCtrl** with the drag operation must make the operation a copy. Pressing **MAlt** with the drag operation must make the operation a move. * * *

Related Information

See Chapter 3 for more information on the drag-and-drop model, and for information on quick transfer and the selection models.

Edit Menu

Description

The common Edit Menu contents are described as follows. Note that you should only include those functions actually supported by your application. The illustration on this reference page shows an Edit Menu.

Undo Alt+Backspace

Must reverse the most recently executed action. To provide a visual cue to the user, the **Undo** selection title should be dynamically modified to indicate what is being undone. For example, if the most recently executed action were a paste, the action name would be **Undo paste**. Your application should be able to undo all of the actions in the **Edit** Menu. This action must have the mnemonic **U**. If this action has an accelerator, it must override the standard undo action for a specific component, and must be **MAlt KBackSpace**. *

Cut Shift+Del

Must remove the selected portion of data from the client area to the clipboard buffer. Your application can determine whether the area that was occupied by the removed data is left blank or whether the remaining data is compressed to fill in the space. Usually, graphics applications leave the space blank while text applications compress the remaining text to fill in the space. This action must have the mnemonic **T**. If this action has an accelerator, it must override the standard cut action for a specific component, and both **MShift KDelete** and **KPrimaryCut** should be supported. *

Copy Ctrl+Ins

Must copy the selected portion of data to the clipboard buffer without removing the original data from the client area. This action must have the mnemonic **C**. If this *

action has an accelerator, it must override the standard copy action for a specific component, and both **MCtrl KInsert** and **KPrimaryCopy** should be supported. *

Paste Shift+Ins

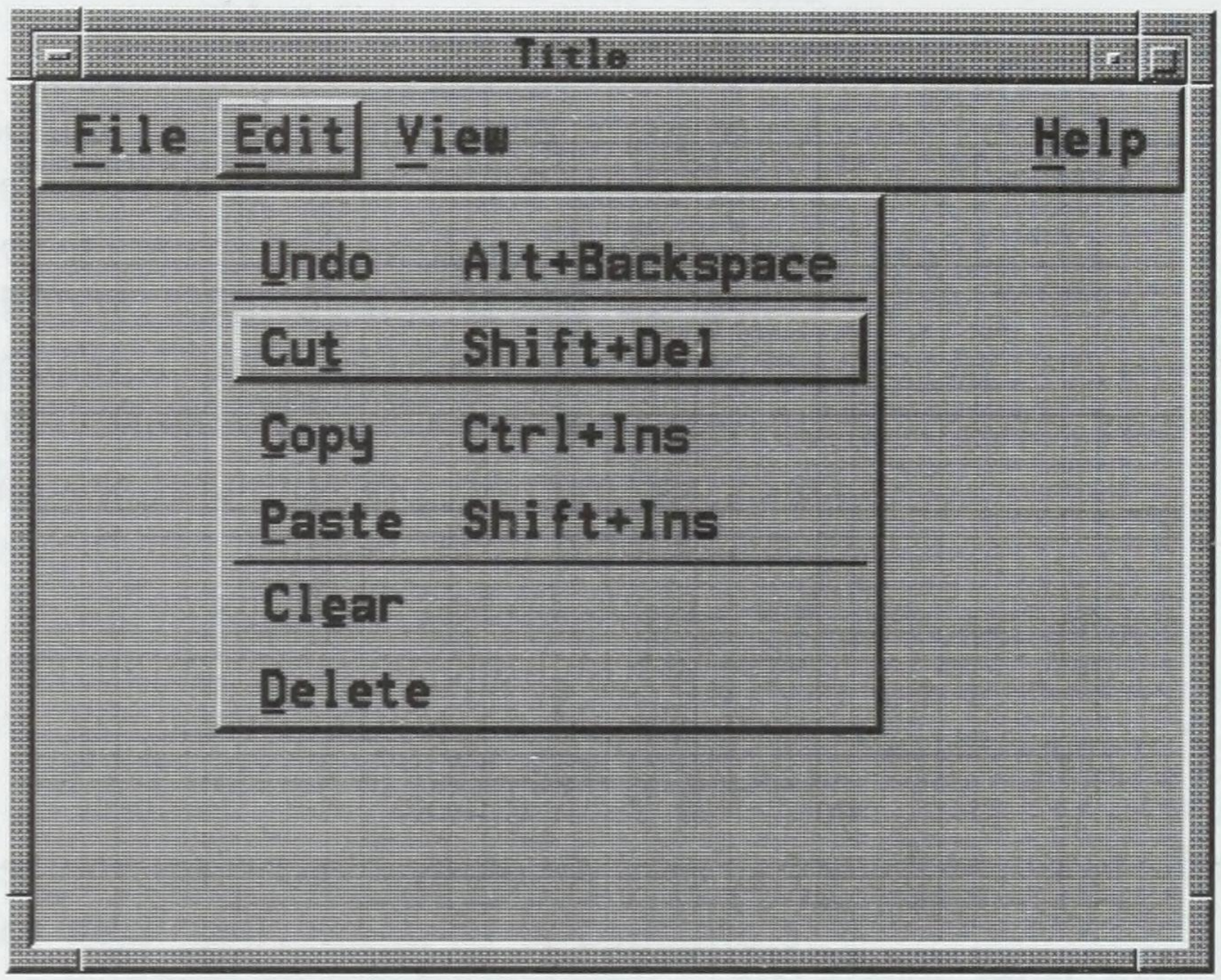
Must paste the contents of the clipboard buffer into a client area. When there is one main editing area, the application can force the paste to occur in that area. When an accelerator is used and the focus is in an editable component the paste should go to that editable component. When the focus policy is explicit, and when there are multiple possible destinations for the paste, the paste should go to the last editable component with the input focus. When the focus policy is implicit, and when there are multiple possible destinations for the paste, the paste should go to the component with the destination cursor, as described in Section 3.2.3. *

Your application can determine whether the pasted data is reformatted to fit in the client area and whether existing data moves to create room for the pasted data. Text applications usually reformat the pasted text to fit into the margins of the text field, and they will move the existing text to make room for the new text. Graphics applications usually do neither. This action must have the mnemonic **P**. If this action has an accelerator, it must override the standard paste action for a specific component, and both **MShift KInsert** and **KPrimaryPaste** should be supported. * * *

Clear Must remove a selected portion of data from the client area without copying it to a clipboard buffer. The remaining data is not compressed to fill the space that was occupied by the cleared data. This action must have the mnemonic **E**. * *

Delete Must remove a selected portion of data from the client area without copying it to a clipboard buffer. The remaining data is compressed to fill the space that was occupied by the deleted data. This action must have the mnemonic **D**. * *

Illustration



Related Information

See Chapter 4 for more information on the MenuBar system, and for general information about Menu design.

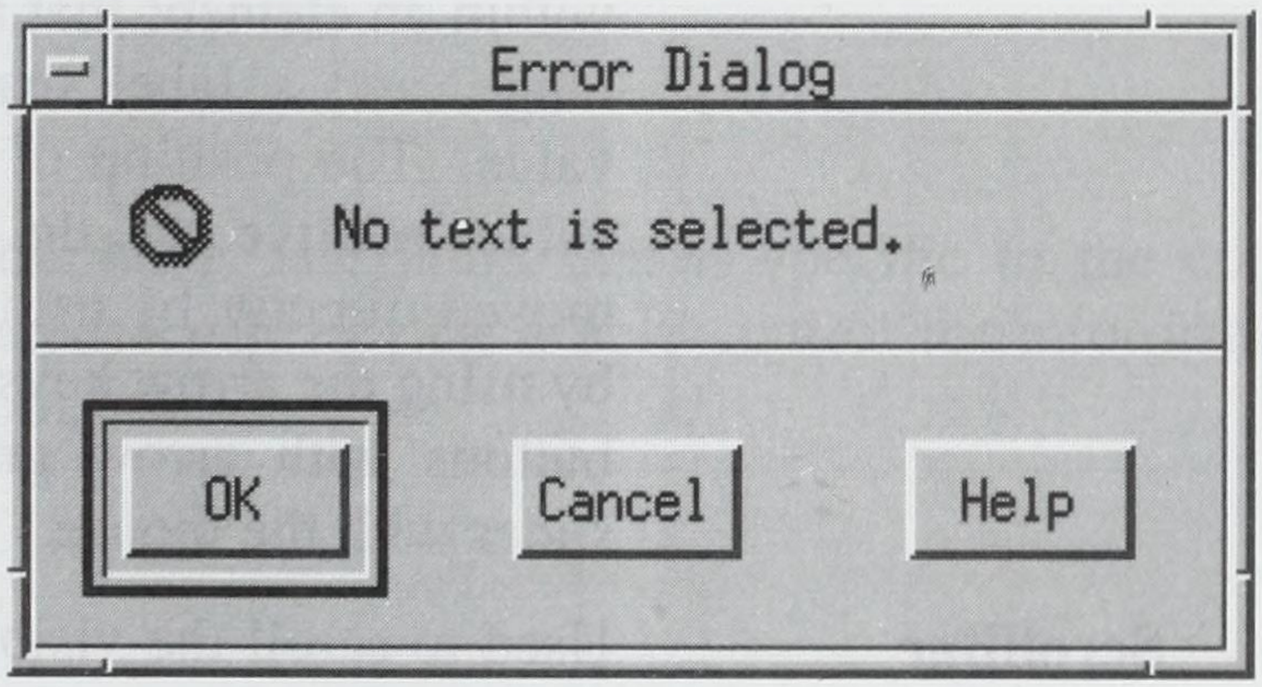
ErrorDialog

Description

An ErrorDialog should be used to convey a message about a user error. It should stop user interaction with the application until it is dismissed. It should include an error symbol, a message, and one of the following button arrangements.

OK Cancel
OK Cancel Help

Illustration



Related Information

For more information, see the reference page for DialogBox.

Field Controls

Description

Field controls are components that use internal navigation controls. Field controls should be navigation fields. The following text describes the field controls.

Sash

Used to set the boundary between two components. A Sash is usually a small square on the boundary between two components. The separated components are called Panes, and a group of Panes, Separators, and Sashes is called a PanedWindow.

Scale

Used to set or display a value in a range. A Scale is usually composed of a slider, moving within an element that indicates the size of the range, and a label that indicates the current value. The position of the slider indicates the value relative to the range. The slider is moved directly by using the mouse pointer, or by using the arrow keys. A Scale can also have buttons with arrow graphics for moving the slider with the mouse.

ScrollBar

Used to scroll the visible area of a component. A ScrollBar is usually composed of a slider, moving within an element that indicates the full size of the component, and buttons with arrow graphics for moving the slider with the mouse. The slider indicates the relative position and size of the visible area of the component. The slider is moved directly by using the mouse pointer, or by using the arrow keys.

List Used for selecting elements from a list of elements. A List can allow multiple items to be selected or can be constrained to only allow one item to be selected at a time. A List is usually composed of a vertical list of items. A List can also have both horizontal and vertical ScrollBars for scrolling the visible portion of the list of items.

Text Used for displaying, entering, and modifying text. There are single line and multiple line variants. Multiline Text can have both horizontal and vertical ScrollBars for scrolling the visible portion of the text area.

Canvas Used for displaying, entering, and modifying graphics. A Canvas can have both horizontal and vertical ScrollBars for scrolling the visible portion of the drawing area.

Navigation

Field controls have navigation that is specific to the component. See the reference page for each field control for information about its navigation.

Other Operations

Field controls have operations that are specific to the component. See the reference page for each field control for information about its operations.

Related Information

For more information, see the reference pages for each field control.

File Menu

Description

The common File Menu contents are described as follows. Note that you should only include those functions actually supported by your application. The illustration on this reference page shows a File Menu.

New Must create a new file. The **New** operation must clear existing data from the client area. If completion of the operation will obliterate current changes to the file, you must display a DialogBox, asking the user about saving changes. This action must have the mnemonic **N**.

Open . . . Must open an existing file. The **Open** operation must prompt the user for the name of the file with a DialogBox. The title bar should be updated with the name of the newly opened file. If completion of the operation will obliterate current changes to the file, you must display a DialogBox, asking the user about saving changes. This DialogBox should be displayed after the user selects a new file. This action must have the mnemonic **O**.

Save Must save the currently opened file without removing the existing contents of the client area. If the currently opened file has no name, **Save** must prompt for a filename with a DialogBox. This action must have the mnemonic **S**.

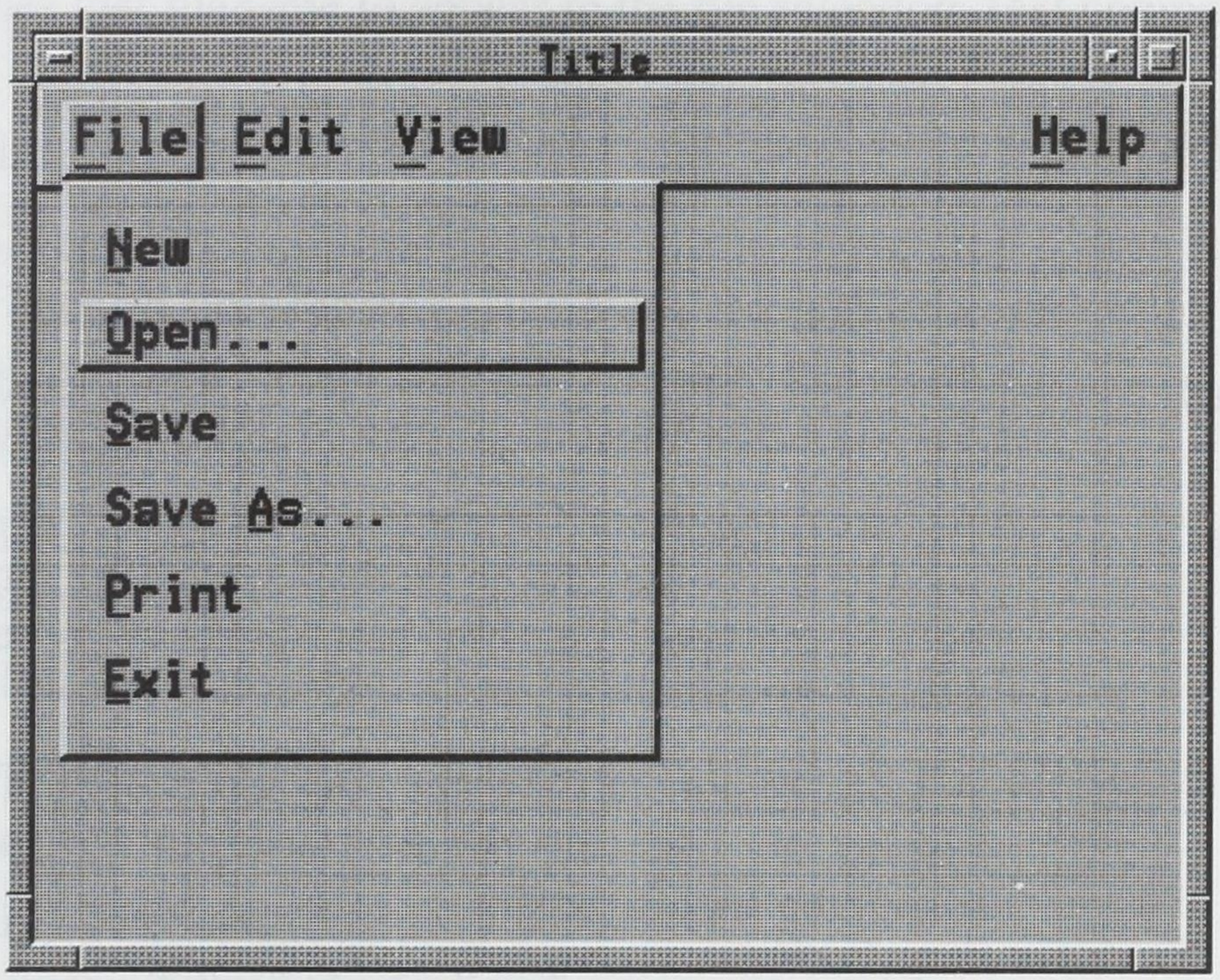
Save As . . . Must save the currently opened file under a new name without removing the existing contents of the client area. The **Save As** operation must prompt for the name of the file with a DialogBox. If the user tries to save the new file under an existing name, **Save As** must alert the user with a DialogBox of a possible loss of data. This action must have the mnemonic **A**.

Print Must schedule a file for printing. If your application requires specific printing information before printing, the operation must first request that information with a DialogBox, and the entry title must be followed by an ellipses. Printing information can also be specified for the application in the **Options** Menu. This action must have the mnemonic **P**.

Close Can be supplied in applications that have multiple independent primary windows. This action must not be supplied in applications with a single primary window, or multiple dependent primary windows. This action must only close the current primary window and its associated secondary windows; that is, the window family. This action must have the mnemonic **C**. You can include this action even though it is similar to the **Close** action in the window Menu. This ensures that users have a way to close the primary window even if they are not running a compliant window manager. Applications should prompt the user to save any unsaved changes if the action would cause loss of data.

Exit Must end the current application and all windows associated with it. This action is equivalent to closing all primary windows of the application. If completion of the operation will obliterate current changes to the file, you must display a DialogBox, asking the user about saving changes. This action must have the mnemonic **E**. You should include this action even though it is similar to the **Close** action in the window Menu. This ensures that users have a way to end the application even if they are not running a compliant window manager. Applications should prompt the user to save any unsaved changes if the action would cause loss of data. If your application does not have a **File** Menu, put **Exit** at the end of the first PullDown Menu.

Illustration



Related Information

See Chapter 4 for more information on the MenuBar system, and for general information about Menu design.

FileSelectionBox

Description

A FileSelectionBox is a special purpose composite component for file selection. It can be used to traverse through directories, view the files and subdirectories in them, and then select files. The FileSelectionBox must be composed of the following elements:

- A Text component for displaying and editing a directory mask used to select the files to be displayed. The directory mask must be a string specifying the base directory to be examined and a search pattern.
- A List component for displaying file names. The file list should display all files and subdirectories in the base directory that match the search pattern.
- A List component for displaying subdirectories. The directory list should display the subdirectories of the base directory, as well as the base directory itself and its parent directory.
- A Text component for displaying and editing a filename.
- A group of PushButtons, labeled **OK**, **Filter**, **Cancel**, and **Help**

The user must be able to select a new directory to examine by scrolling through the list of directories and selecting the desired directory or by editing the directory mask. Selecting a new directory from the directory list must not change the search pattern. A user must be able to select a new search pattern by editing the directory mask. Double clicking **BSelect** or pressing **KActivate** on a directory in the directory list must initiate a search for files and subdirectories in the new directory using the current search pattern.

The user must be able to select a file by scrolling through the list of filenames and selecting the desired file or by entering the filename directly into the Text component. Selecting a file from the list causes that filename to appear in the file selection Text component.

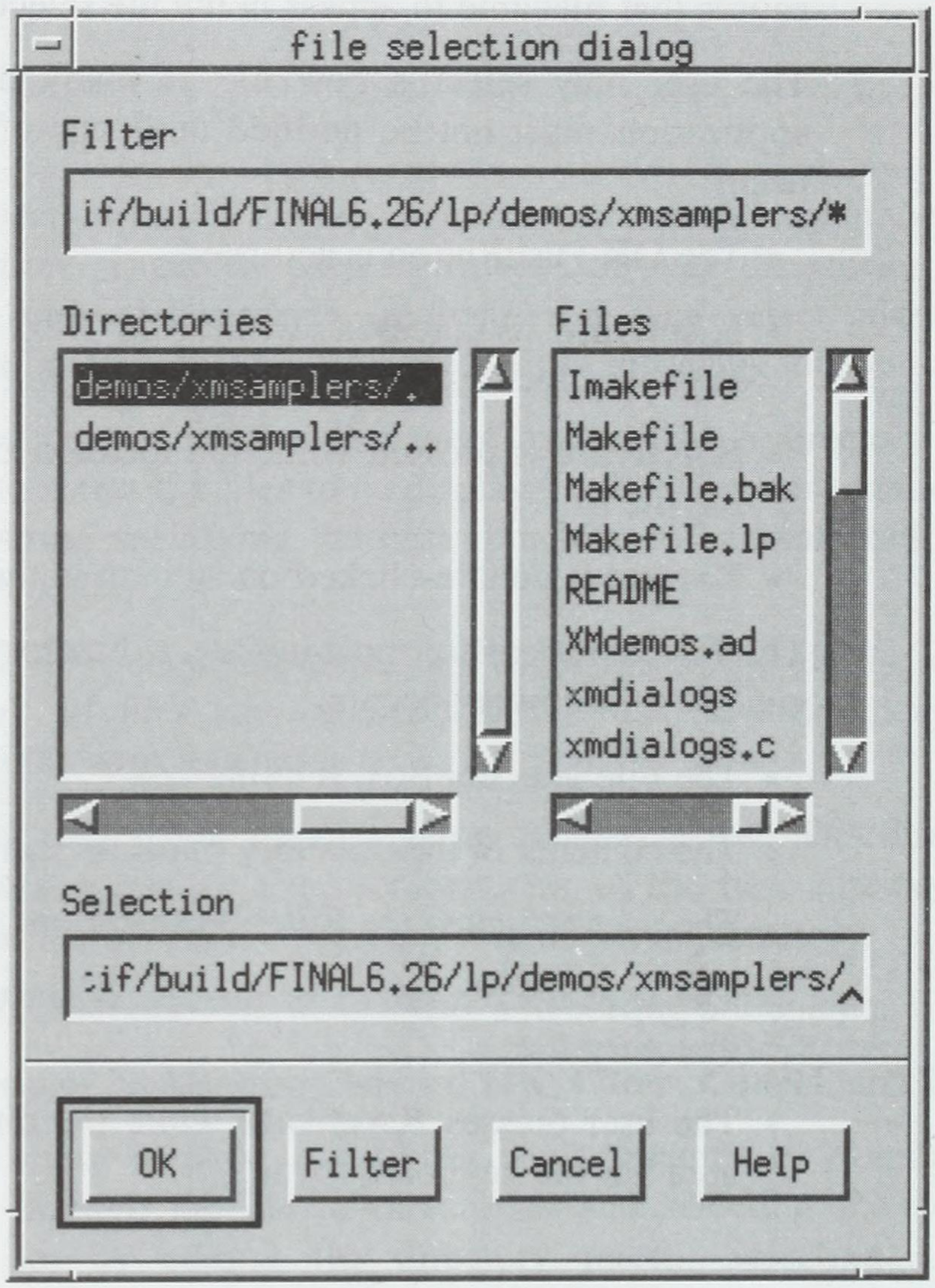
The user may select a new file as many times as desired. The application must not be notified until one of the following events occur:

- The **OK** PushButton is activated.
- **KActivate** is pressed while the Text component has the keyboard focus.
- **KActivate** is pressed while the location cursor is on an item in the file list.
- **BSelect** is double-clicked on an item in the file list.

The FileSelectionBox must initiate a directory and file search when any of the following occurs:

- The FileSelectionBox is initialized
- The contents of the directory mask is changed
- The user activates the **Filter** PushButton
- The user double-clicks or presses **KActivate** on an item in the directory list
- The user presses **KActivate** while the directory mask text edit area has the keyboard focus

Illustration



Related Information

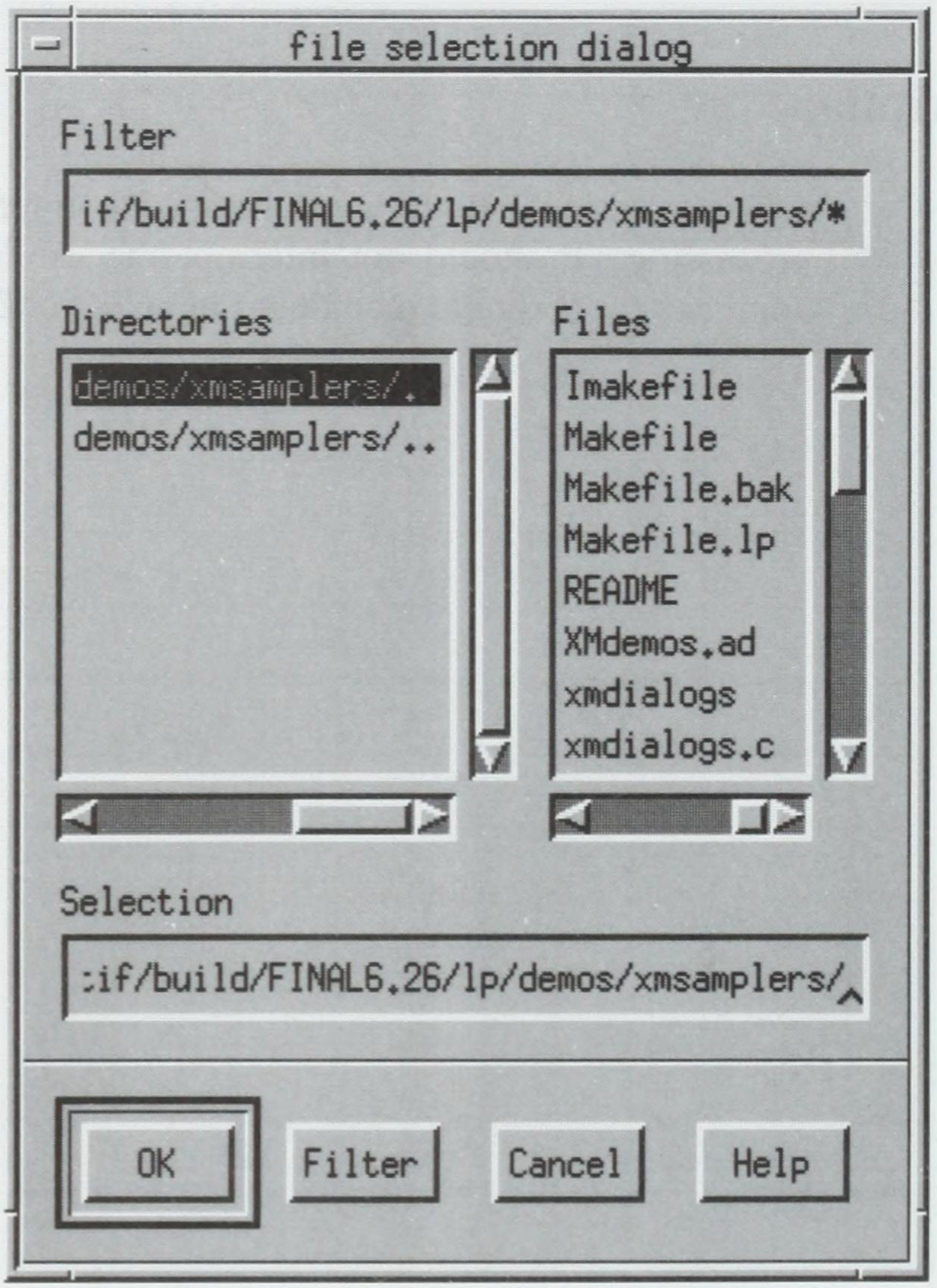
For more information, see the reference page for FileSelection DialogBox.

FileSelectionDialog

Description

A FileSelectionDialog should be used to enter the name of a file for processing. It should not interrupt the user's interaction with the application. It should include a FileSelectionBox.

Illustration



Related Information

For more information, see the reference page for DialogBox.

Focus

Description

The keyboard focus model is defined by a focus policy. A focus policy is a specific mechanism for moving the focus among windows and components. The implicit and explicit focus policies must be supported. *

In the implicit focus policy, also called the pointer, real-estate driven, or point-to-type policy, the keyboard focus moves to the window or component into which a user moves the mouse pointer. No explicit action is performed to set the keyboard focus in the implicit focus model. Keyboard events are sent to the window or component that the mouse pointer is in, more specifically to the component that the mouse pointer is in. In implicit mode, the keyboard focus tracks the mouse pointer. Because of this, there is no way to move the keyboard focus from the keyboard using implicit mode. In this focus policy, the location cursor for keyboard events does not need to be shown; however, the application can show it.

The mouse pointer always behaves as if the focus policy is implicit. That is, mouse events always go to the window and the component that the mouse pointer is in.

The explicit focus policy requires the user to explicitly select which window or component receives the keyboard focus. Mouse events are sent to the component that the pointer is over, so the mouse pointer must always behave as if the focus policy is implicit, even when the focus policy is explicit. In explicit focus mode, a user moves the keyboard focus to a window or component by pressing **BSelect** while the mouse pointer is over a window or component. Simply moving the mouse pointer over a window or component does not give the window or component the keyboard focus. In this focus policy, the location of keyboard focus must be shown by a location cursor. * * * * * * *

In explicit mode, keyboard focus moves explicitly among the components in a window as well as among the windows. Moving the keyboard focus among components in a window is called window navigation. Keyboard events go to the component in the window with the keyboard focus.

Related Information

See Chapter 2 for more information about the focus model.

Frame

Description

A Frame should be used to frame other components. It simply provides a decorative border.

Other Operations

This component can indicate the input focus.

Framing Groups

Description

Framing groups are used to frame groups of components as the following text describes.

Frame	Draws framing decorations around a component.
ScrolledWindow	Frames a component and adds ScrollBars for scrolling the visible area of the component.
MainWindow	Organizes the contents of a primary window. A MainWindow frames the client area and can optionally include ScrollBars, a MenuBar, a command area, and a message area.

Navigation

These groups must follow the navigation model as described in * Chapter 2.

Related Information

For more information, see the reference pages for each framing group.

Help Menu

Description

The common Help Menu contents are described as follows. Note that you should only include those functions actually supported by your application. The illustration on this reference page shows a Help Menu.

On Context Must initiate context-sensitive help by changing the shape of the pointer to the question pointer described in Section 2.2.2. When the user moves the pointer to the component help is wanted on and presses **BSelect**, any available context-sensitive help for the component must be presented, and the pointer reverts from the question pointer. This action must have the mnemonic **C**. *

On Help Must provide information on how to use the application's help facility. *

On Window Must provide general information about the window from which help was requested. *

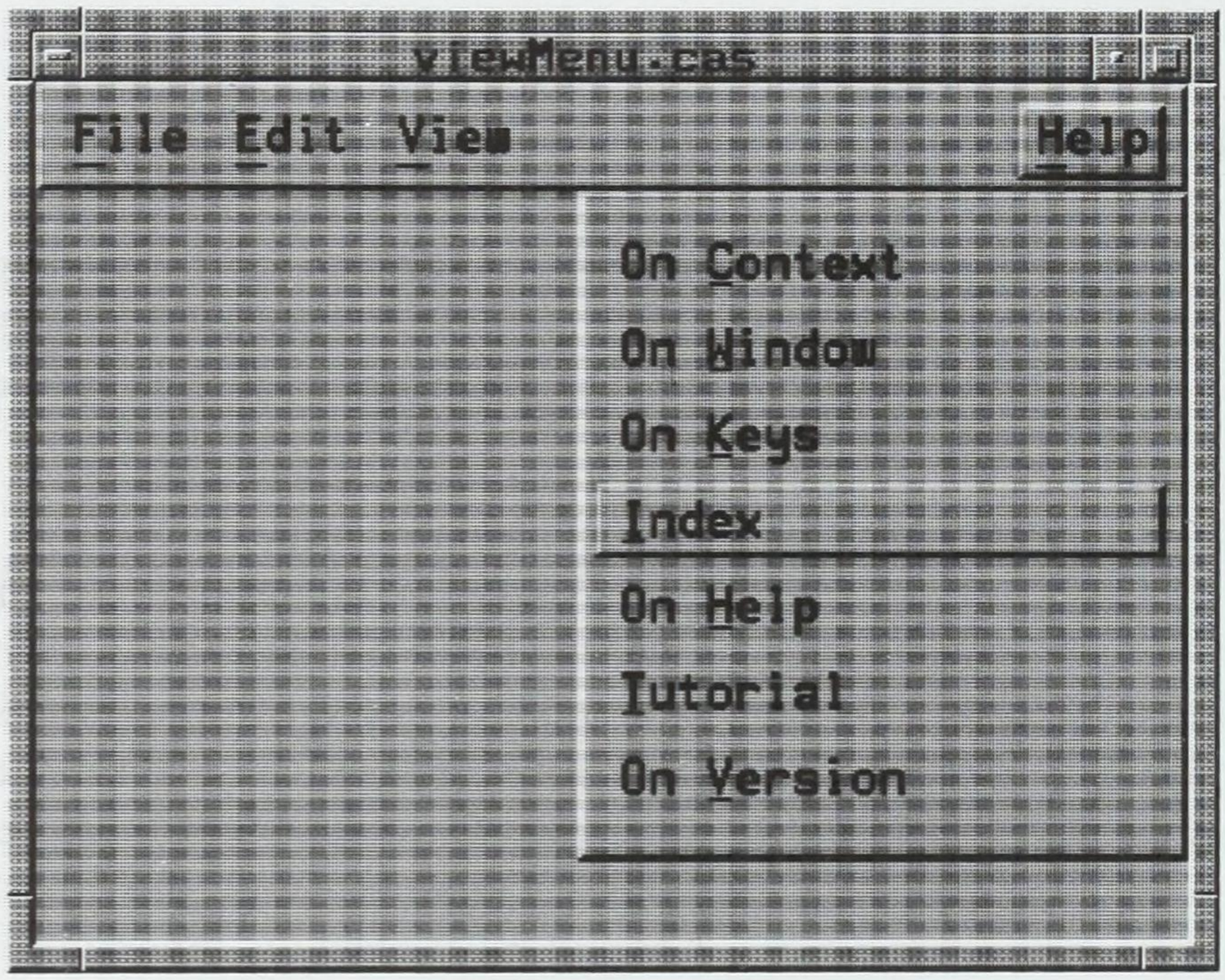
On Keys Must provide information about the application's use of function keys, mnemonics, and keyboard accelerators. *

Index Must provide an index for all help information in the application. The index can provide search capabilities. *

Tutorial Must provide access to the application's tutorial. *

On Version Must provide the name and version of the application. It can provide other information as well. *

Illustration



Related Information

See Chapter 4 for more information on the MenuBar system, and for general information about Menu design.

Icon Menu

Description

Clicking **BSelect** in an icon must give the icon keyboard focus and post the icon Menu. The icon Menu must be the same Menu as the window Menu for the associated primary window. **Size** and **Minimize** must not be available from the icon Menu. *

Double-clicking **BSelect** anywhere in the icon must restore the window family, just as the icon Menu item **Restore** does. Pressing **BSelect** anywhere in the icon and dragging the mouse pointer must move the icon to track the pointer, just as the icon Menu item **Move** does. *

Related Information

See Chapter 5 for more information on the icon Menu and the window manager in general. See Chapter 4 for information about Menu design.

IconBox

Description

An IconBox is a specialized window that acts as a storage location for icons. The illustration on this reference page shows a typical IconBox.

The window manager must allow the IconBox to have all the same components as any other primary window. Like other windows, it can be sized, moved, minimized, maximized, restored, and lowered. However, the window manager must not allow the IconBox to be closed. The system Menu action **Close** must be replaced with the system Menu action **Pack Icons** in an IconBox. **Pack Icons** arranges the icons as close as possible together in the visible icon area. **Pack Icons** must have the mnemonic **P**. If **Pack Icons** has an accelerator, it must be **MAlt <F12>** if **<F12>** is available.

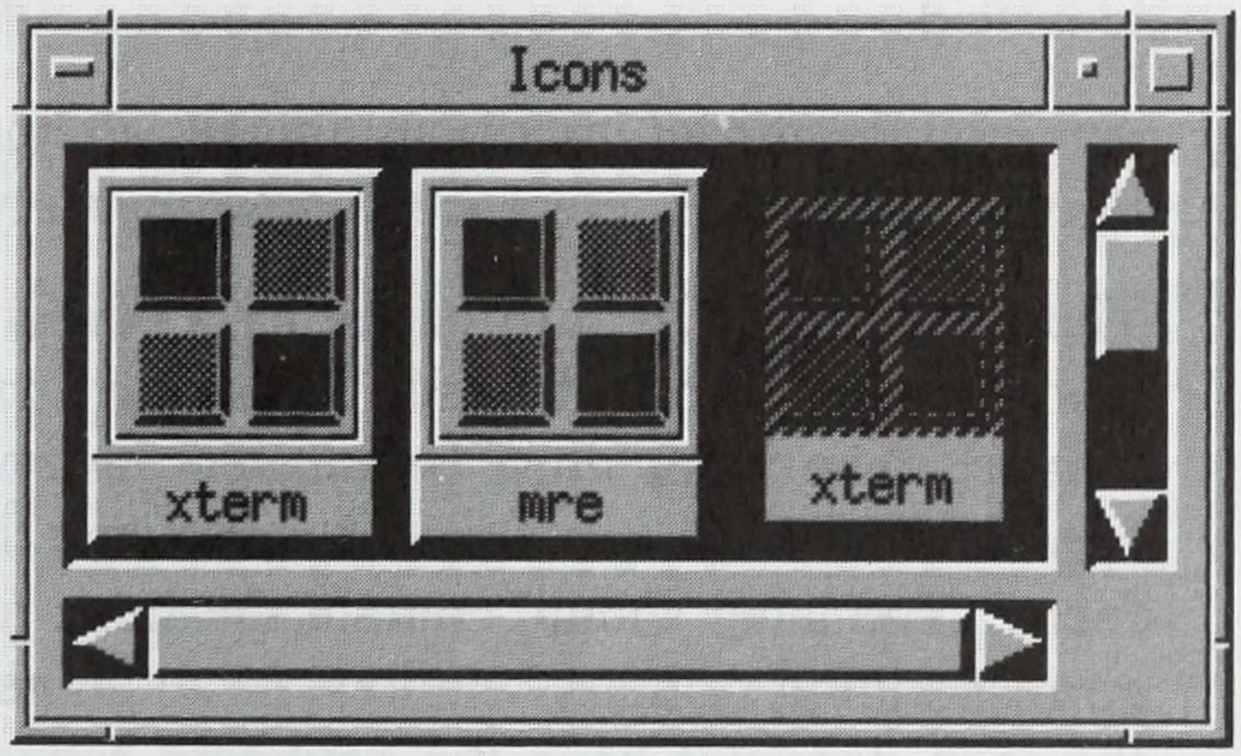
The IconBox can contain an icon for each window family, even if it is active. The icon for minimized window families must be the same as the icon would be outside of the IconBox. The icon for an active window family should be similar to the minimized icon, deemphasized somehow, and without the icon image border.

Clicking **BSelect** in an icon must give the icon keyboard focus and can post the icon Menu. The icon Menu must be the same Menu as the window Menu for the associated primary window. If the icon represents an active window family, **Restore**, **Size**, **Minimize**, and **Maximize** must not be available from the icon Menu. If the icon represents a minimized window family, **Size** and **Minimize** must not be available from the icon Menu.

Double-clicking **BSelect** anywhere in an icon that represents a minimized window family must restore the window family, just as

the icon Menu item **Restore**. Double-clicking **BSelect** anywhere in an icon that represents an active window family must raise the window family to the top of the window hierarchy. Pressing **BSelect** or **BDrag** anywhere in the icon and dragging the mouse pointer must move the icon within the icon area to track the pointer, just as the icon Menu item **Move** does.

Illustration



Related Information

See Chapter 5 for more information about icons and the window manager.

Icons

Description

A window icon is a minimized representation of a window or window family that can help organize windows and tasks in the display. The window manager must iconify all windows of a * window family together. The application running inside of a * window must continue running even when the application is * iconified.

An icon is made up of an image, a border, and a label. The illustration on this reference page shows a typical Motif icon.

The image must be surrounded by a border that indicates when the * icon has the keyboard focus. The image area can contain text or a bitmap. The label must be located just below the image and its * border. The label can also indicate when the icon has the keyboard focus. The label should contain the same text as the title bar of the corresponding primary window, or an abbreviated form of it. When * the icon does not have the keyboard focus, the width of the label * must be the same as the width of the image area and its border. The * text must be centered in the label unless it is too large to fit in the * label, in which case the text should be left justified in the label and * any extra text truncated. When the icon has the keyboard focus, the width of the label can expand to display the entire text. The * expanded label must be centered on the icon unless it would push * off the screen, in which case the label should be as close to center as is possible while keeping the entire label visible.

Clicking **BSelect** in an icon must give the icon keyboard focus and * post the icon Menu. Double-clicking **BSelect** anywhere in the icon * must restore the window family, just as the icon Menu item * **Restore**. Pressing **BSelect** anywhere in the icon and dragging the * mouse pointer must move the icon to track the pointer, just as the * icon Menu item **Move**.

Illustration

Related Information

See Chapter 5 for more information about icons and the window manager.

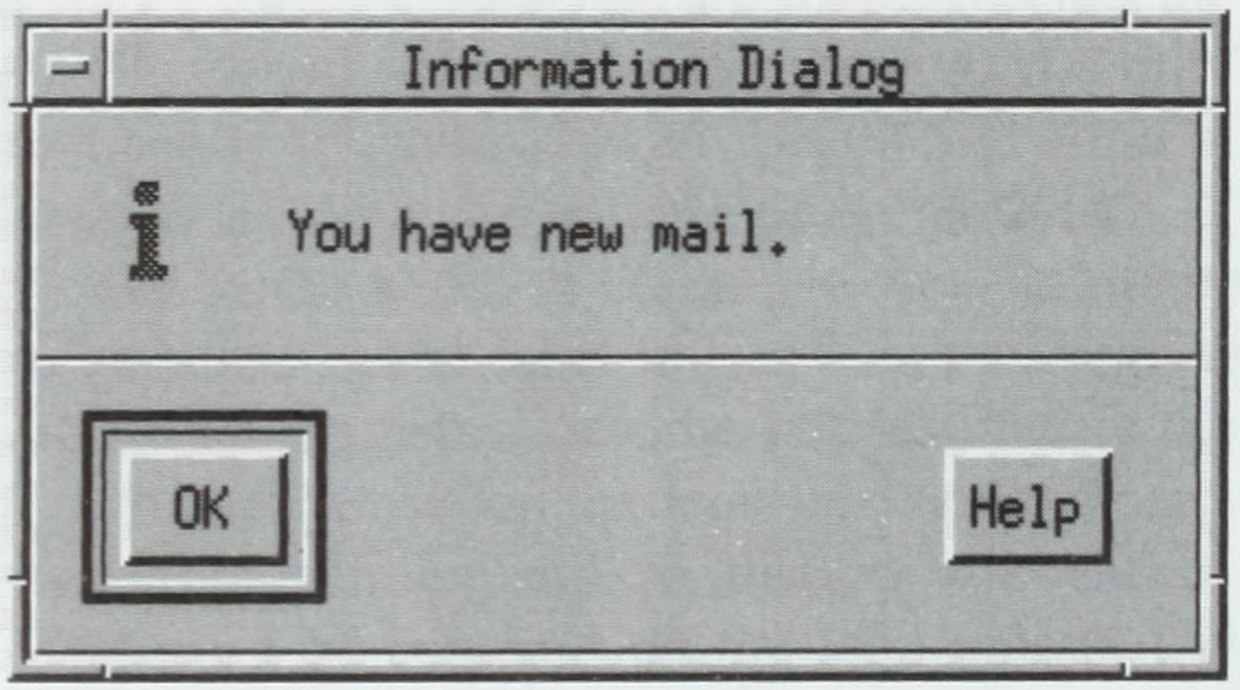
InformationDialog

Description

An InformationDialog should be used to convey information to the user. It must not interrupt the user's interaction with the application. It should include an information symbol, a message, and one of the following button arrangements. *

OK
OK Help

Illustration



Related Information

For more information, see the reference page for DialogBox.

Input Devices

Description

The most typical pointing device is a mouse, although a graphics tablet, track ball, joystick, and other tools also work as pointing devices. You can use any pointing device in place of a mouse. This guide assumes that a mouse, or any pointing device, has the following three buttons. Chapter 2 describes the button bindings for pointing devices that do not have three buttons.

BSelect Used for selection, activation, and setting the location cursor. This button must be the leftmost mouse button, * except for left-handed users where it can be the * rightmost button.

BDrag Used for moving and copying elements. This button * must be the middle mouse button.

BCustom Used for additional user interactions, including popping up Menus and special dialogs, and for application-specific interactions. This button must be the rightmost * mouse button, except for left-handed users where it can * be the leftmost button. This guide also mentions the * button **BMenu**, which must be bound to **BCustom**. **BMenu** is used for popping up Menus.

Because not all keyboards are the same, it is difficult to give style guidelines that are correct for every manufacturer's keyboard. To solve this problem, this guide describes keys using a virtual mechanism; that is, by the action they perform. Everywhere that keyboard input is specified, the keys are indicated by their virtual key name.

The virtual key mechanism includes keys, modifiers, and mouse buttons. The mouse button virtual keys are described earlier in this reference page as **BSelect**, **BDrag**, **BCustom**, and **BMenu**. Note

that all button virtual keys start with the letter **B**. All normal key virtual keys start with the letter **K**. All modifier key virtual keys start with the letter **M**.

This guide assumes that a keyboard has all the standard alphabetic, numeric, and symbol keys, as well as the special keys **<Return>**, **<Tab>**, **<Escape>**, **<Delete>**, **<Spacebar>**, and **<Backspace>**. This *Style Guide* describes the common bindings of some actions in terms of the function keys **<F1>** through **<F10>**. These are nominal bindings and can be changed if the function keys are not available, or if another binding is preferred. It also assumes that the arrow keys are available. Additional keys, including navigation keys and special keys like **<Menu>** and **<Help>**, are suggested, but not necessary for full keyboard equivalence.

This guide also assumes that all keyboards have the three modifier keys **MShift**, **MCtrl**, and **MAlt**, which are usually bound to **<Shift>**, **<Ctrl>**, and **<Alt>** or **<Meta>**. Your keyboard may have the virtual buttons, keys, or modifiers bound as appropriate. Appendix A lists all of the virtual keys used in this guide, as well as suggested and alternative bindings.

Related Information

See Chapter 2 for a more information about input devices. See Appendix A for a listing of virtual keys and their common bindings.

Label

Description

A Label should be used to display text or graphics that label other components. This component must be composed of an area for displaying a text or graphics label. *

Illustration

Navigation

A Label must have no internal navigation. *

Layout Groups

Description

Layout groups are used for organizing components into groups as described in the following text.

Composition Organizes a collection of components, including groups, in an arbitrary layout.

PanedWindow A linear grouping of components and Sashes. Sashes are used to set the boundary between two components. The separated components are called Panes and can contain any components.

Navigation

Layout groups must follow the navigation model as described in * Chapter 2.

Related Information

For more information, see the reference pages for each layout group.

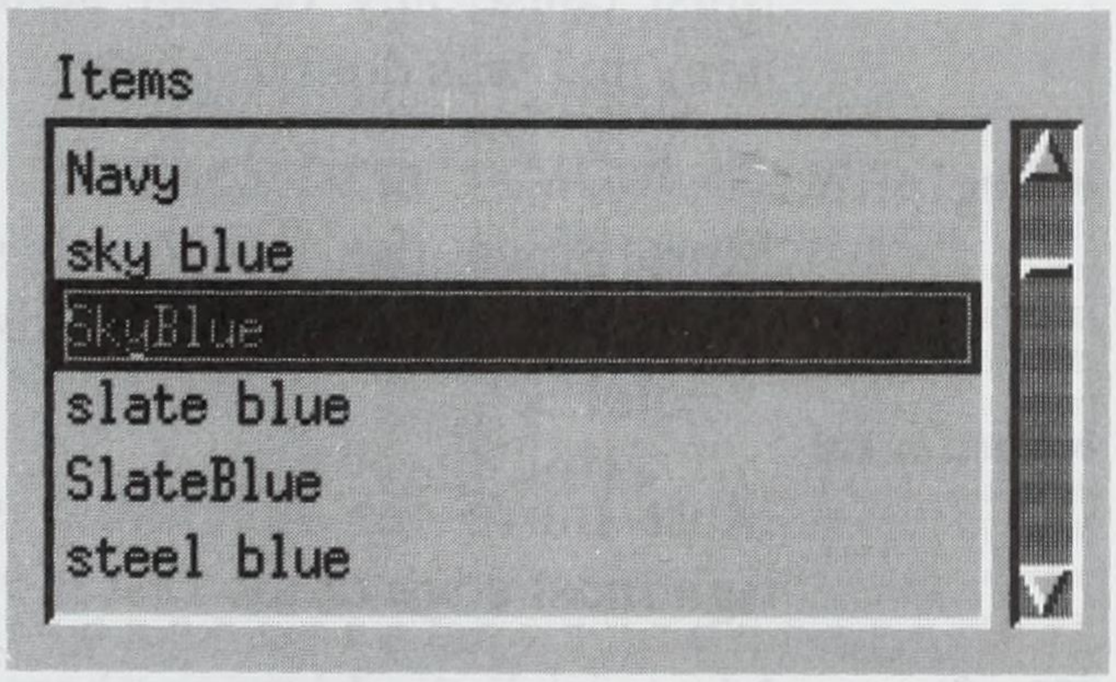
List

Description

A List should be used to present a list of elements for selection. The List elements can be selected using either the single selection model, the browse selection model, the multiple section model, or the discontiguous selection model. This component must be composed of an area for presenting a list of text or graphics elements. It can optionally have vertical and horizontal ScrollBars, which show different views of the List elements.

The location cursor in a List should differentiate between normal mode and add mode. The location cursor should be a solid box in normal mode and a dashed box in add mode.

Illustration



Navigation

The following text describes the navigation actions of this component.

KBeginData
Must move the location cursor to the first item in the List. *

KEndData
Must move the location cursor to the last item in the List. *

KUp Must move the location cursor to the previous item in the List. *

KDown Must move the location cursor to the next item in the List. *

Other Operations

The following text describes the operations of this component.

BSelect Click 2
Must cause any default action for the List using the item that was double-clicked as the selection. *

KBeginLine
Must move the horizontal scroll region so that the leftmost edge of the List is visible. *

KEndLine
Must move the horizontal scroll region so that the rightmost edge of the List is visible. *

KLeft Must scroll the List one character to the left. *

KPageLeft
Must scroll the List one page to the left. *

KPageDown
Must scroll the List to the next page. *

KPageUp Must scroll the List to the previous page. *

KRight Must scroll the List one character to the right. *

KPageRight
Must scroll the List one page to the right. *

KHelp Must provide any available help for the List. *

Common Bindings

Virtual Keys	Default Bindings
BSelect	<Btn1>
KUp	<↑>
KDown	<↓>
KLeft	<←>
KRight	<→>
KPageUp	<PageUp>
KPageDown	<PageDown>
KPageLeft	MCtrl <PageUp>
KPageRight	MCtrl <PageDown>
KBeginLine	<Begin>
	<Home>
KEndLine	<End>
KBeginData	MCtrl <Begin>
	MCtrl <Home>
KEndData	MCtrl <End>
KHelp	<Help>
	<F1>

Related Information

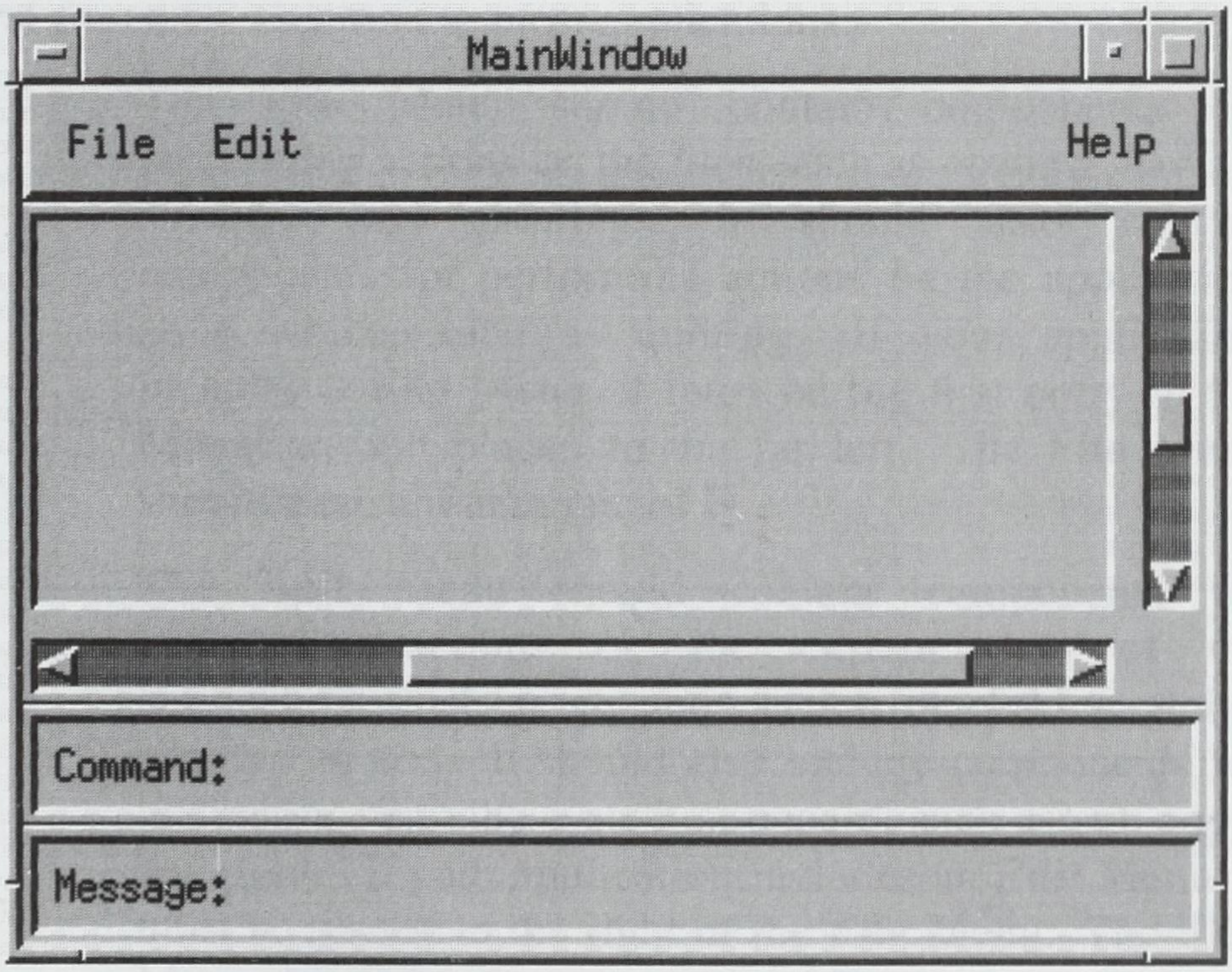
For more information on List element selection, see the description of the selection models in Chapter 3. For more information about the ScrollBars, see the reference page for ScrollBars.

MainWindow

Description

A MainWindow should be used to organize the contents of a primary window. A MainWindow must frame the client area and * can optionally include ScrollBars, a MenuBar, a command area, and a message area.

Illustration



Navigation

A MainWindow must follow the navigation model described in * Chapter 2.

Related Information

For more information about the components of the MainWindow, see the reference pages for MenuBar, ScrollBar, and Text.

MenuBar

Description

A MenuBar should be used to organize the most common features of an application. The MenuBar must be a horizontal bar at the top edge of the application just below the title area of the window frame. The MenuBar contains a list of Menu topics in CascadeButtons with PullDown Menus connected to them. Commands must not be included as topics in the MenuBar because they inhibit Menu browsing. Each Menu topic should have a single letter mnemonic indicated by an underline.

The following Menus pulled down from the MenuBar provide general functions common to most applications.

File The **File** Menu should contain components for performing actions on the files, such as opening, saving, closing, and printing. It should also contain components for performing actions on the application as a whole, such as quitting. If your application includes a **File** Menu, it must be the first entry in the MenuBar, and placed to the far left. The **File** Menu should have a mnemonic of **F**.

Edit The **Edit** Menu should contain components for performing actions on the current data of the application, such as an undo action or making global changes to text. It should also include components for interacting with the system clipboard, such as cut, copy, and paste. If your application includes an **Edit** Menu, it must be placed to the right of the **File** Menu. The **Edit** Menu should have a mnemonic of **E**.

View The **View** Menu should contain components for changing the users view on the data. Components in the **View** Menu should not actually change the data. The exact contents of the **View** Menu are application specific, but it may include components that change the

appearance of the data, the amount of data that is displayed, or the order in which the data is displayed. If your application includes a **View** Menu, it must be placed to the right of the **Edit** Menu. The **View** Menu should have a mnemonic of **V**. *

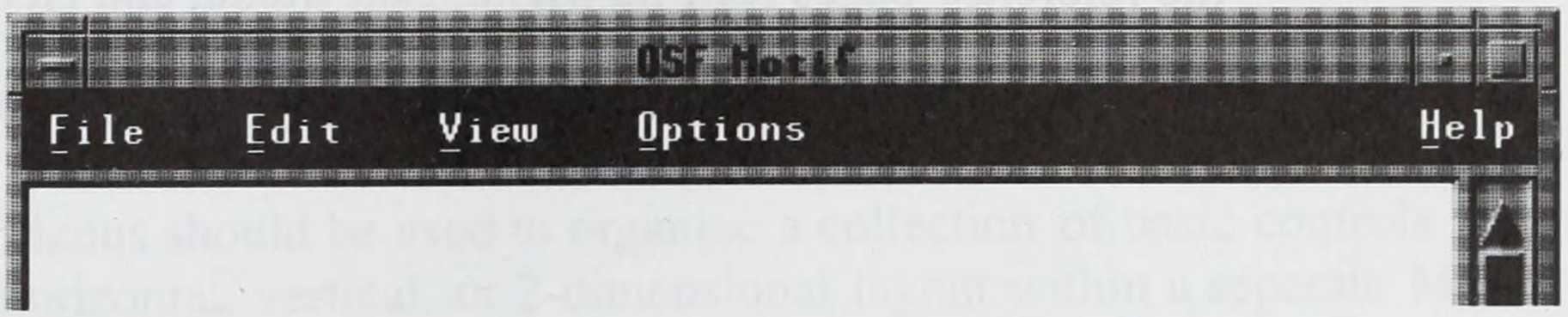
Options The **Options** Menu should contain components for a user to customize the application. The exact contents of the **Options** Menu are application specific. If your application includes an **Options** Menu, it must be placed to the right of the **View** Menu. The **Options** Menu should have a mnemonic of **O**. *

Help The **Help** Menu should contain components that provide user help facilities. The components in the **Help** Menu usually bring up a DialogBox with help information. Every application should have a **Help** Menu. If your application includes a **Help** Menu, it must be the last Menu item, and placed to the far right on the MenuBar. The **Help** Menu should have a mnemonic of **H**. *

While we recommended that you include the common Menus in the MenuBar of your application, your choice of Menu titles and items depends on the nature of your application. If your application requires it, you should design more relevant titles and selections, but do not change the meanings of words used in the common Menus.

The recommended contents of the **File**, **Edit**, and **Help** Menus are described on their own reference pages. The contents of the **View** and **Options** Menus are application specific and are not specified here.

Illustration



Navigation

The MenuBar follows the navigation model described in Chapter 2.

Other Operations

The following text describes the operations of this component.

KHelp Must provide any available help for the current CascadeButton. *

Common Bindings

Virtual Keys	Default Bindings
KHelp	<Help> <F1>

Related Information

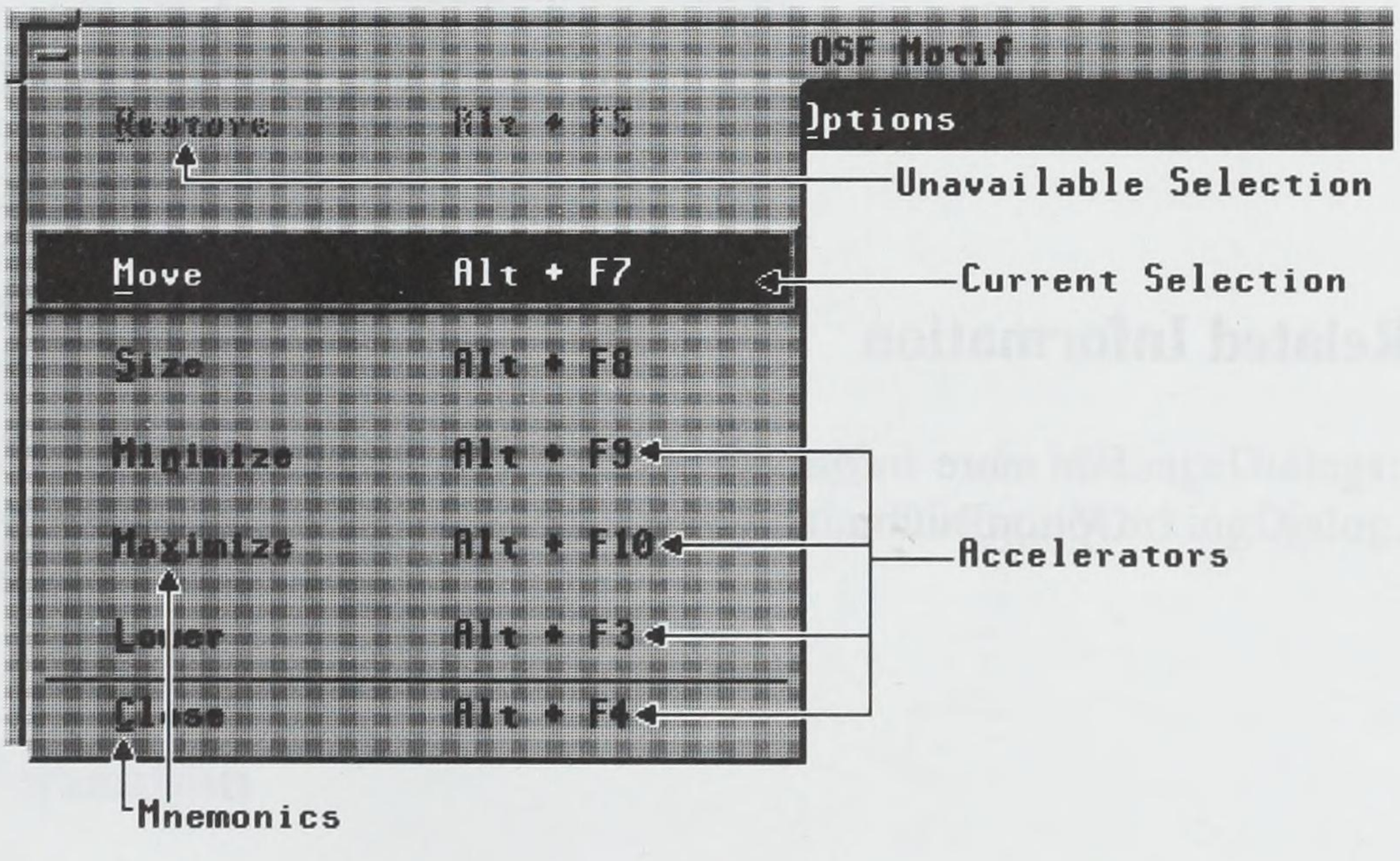
For information about the suggested contents of these Menus, see the reference pages for File Menu, Edit Menu, and Help Menu.

Menus

Description

Menus should be used to organize a collection of basic controls in a horizontal, vertical, or 2-dimensional layout within a separate Menu window. There are three types of Menus: PullDown, Popup, and Option. A Menu is only available while it is posted.

Illustration



Navigation

Menus follow the navigation model described in Chapter 2.

Other Operations

The following text describes the operations of this component.

KHelp Must provide any available help for the Menu. *

Common Bindings

Virtual Key	Default Bindings
KHelp	<Help> <F1>

Related Information

For more information, see the reference pages for CascadeButton, OptionButton, and MenuBar.

MessageDialogs

Description

MessageDialogs should be used to convey a message to the user. * They must not interrupt the user's interaction with the application. They should include a message, and one of the following button arrangements.

OK
OK Help
OK Cancel
OK Cancel Help
Yes No
Yes No Help
Yes No Cancel
Yes No Cancel Help
Cancel
Cancel Help
Retry Cancel
Retry Cancel Help

There are a number of different types of MessageDialogs: ErrorDialog, InformationDialog, QuestionDialog, WorkingDialog, and WarningDialog.

Related Information

For more information, see the reference pages for DialogBox, ErrorDialog, InformationDialog, QuestionDialog, WorkingDialog, and WarningDialog.

Mnemonics

Description

A mnemonic is a single character that can be associated with any component that contains a text label. The label must contain the * character, and the character must be underlined within the label. The mnemonic should be the first character of the label. If a label * does not naturally contain the character of the mnemonic, the * mnemonic must be placed in parentheses following the label. Labels can also be sequentially numbered, and the number can serve as the mnemonic. Labels that are duplicated within an application should be given the same mnemonic. Mnemonics must be case * insensitive for activation. Either an uppercase or lowercase letter can be underlined in the label.

When the location cursor is on a component within the same field as * a component with a mnemonic, a Menu, or a MenuBar, typing the * mnemonic character must move the location cursor to the * component and activate it. If a mnemonic is used for an * OptionButton or for a CascadeButton in a MenuBar, or optionally * for a PushButton that is not in a basic group, typing **MAlt** and the * mnemonic anywhere in the window or its Menus must move the * cursor to the component with that mnemonic and must activate it.

Collections can also support mnemonics associated with its elements. Typing a mnemonic while the collection has the * keyboard focus must be equivalent to moving the location cursor to * the element and typing **KSelect**.

Navigation

Description

Using a mouse, navigation is a simple matter of moving the mouse pointer. Using the keyboard, the following navigation actions must be available. *

KNextField Must move the location cursor forward through fields. If the next field is a field control, **KNextField** must move the location cursor to it. If the next field is a group, **KNextField** must move the location cursor to the first traversable component within the group, unless the field contains a default component. If the field contains a default component, **KNextField** must move the location cursor to the default component instead of the first component. If the next field contains no traversable components, **KNextField** must skip the field. *

KPrevField Must move the location cursor backward through fields using the same rules as **KNextField**. *

KDown In a left-to-right direction language environment, **KDown** must move the location cursor through all traversable components in the group, starting at the upper-left component and ending at the lower-right component, then wrapping back up to the upper-left component. In a right-to-left direction language environment, the location cursor can move, starting at the upper-right component and moving to the lower-left component. *

KUp Must move the location cursor through all traversable components in the opposite direction of **KDown**. *

KRight Must move the location cursor through all traversable components in the group, starting at the upper-left component and ending at the lower-right component, then wrapping back up to the upper-left component. *

KLeft Must move the location cursor through all traversable components in the opposite direction of **KRight**. *

KMenuBar If the MenuBar is inactive, **KMenuBar** must traverse to the MenuBar system. The location cursor must be placed on the first CascadeButton that accepts input in the MenuBar. If there are no CascadeButtons that accept input in the MenuBar, **KMenuBar** must do nothing. If the MenuBar system is active, **KMenuBar** must unpost all Menus in the MenuBar system and return the location cursor to the component that previously had it, if the focus policy is explicit. *

KMenu If the keyboard focus is in an area with an inactive Popup Menu, **KMenu** must pop up the Menu. The location cursor must be placed on the first traversable item in the Popup Menu. If there are no traversable items in the Popup Menu, this has no effect. If there is an active Popup Menu, **KMenu** must unpost the Popup Menu system and return the location cursor to the component that previously had it, if the focus policy is explicit. *

KNextFamilyWindow
Must move the focus to the next window in the window family. *

KPrevFamilyWindow
Must move the focus to the previous window in the window family. *

KNextWindow Must move the focus to the last window that had the input focus in the next window family in the window hierarchy. *

KPrevWindow Must move the focus to previous last window that had the input focus in the next window family in the window hierarchy. *

Common Bindings

Virtual Key	Default Bindings
KNextField	<Tab>
	MCtrl <Tab>
KPrevField	MShift <Tab>
	MCtrl MShift <Tab>
KDown	<↓>
KLeft	<←>
KRight	<→>
KUp	<↑>
KMenuBar	<F10>
KMenu	<F4>
KNextFamilyWindow	MAlt <F6>
KPrevFamilyWindow	MShift MAlt <F6>
KNextWindow	MAlt <Tab>
KPrevWindow	MAlt MShift <Tab>

Related Information

See Chapter 2 for more information about internal window navigation. See Chapter 5 for more information about navigation among windows.

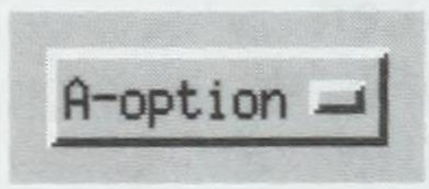
OptionButton

Description

An OptionButton should be used to post an Option Menu.

This component must be composed of a button, with either a text or graphics label. Following the label, this component should also include a bar graphic to distinguish it from a PushButton. The label must be the last selection made from the OptionButton. *

Illustration



Navigation

An OptionButton must have no internal navigation. *

Other Operations

The following text describes the operations of this component.

BSelect Press

Must activate the OptionButton and post the associated Option Menu. *

BSelect Release

If the release is within the same OptionButton as the press, **BSelect Release** must deactivate the OptionButton and leave the associated Option Menu posted. If the release is outside of the OptionButton, **BSelect Release** must deactivate the OptionButton and unpost the associated Option Menu. * * * * *

KActivate Must post the associated Option Menu. *

KSelect Must post the associated Option Menu. *

MAny KCancel

Must unpost the last posted Menu in the Menu system without changing the value of the OptionButton. * *

KHelp Must provide any available help for the OptionButton. *

Common Bindings

Virtual Keys	Default Bindings
BSelect	<Btn1>
KEnter	<Return>
	<Enter>
KActivate	KEnter
	MCtrl KEnter
KSelect	<Select>
	<Space>
	MCtrl <Space>
KHelp	<Help>
	<F1>
KCancel	<Escape>
	<Cancel>

Related Information

For more information, see the reference page for Menus.

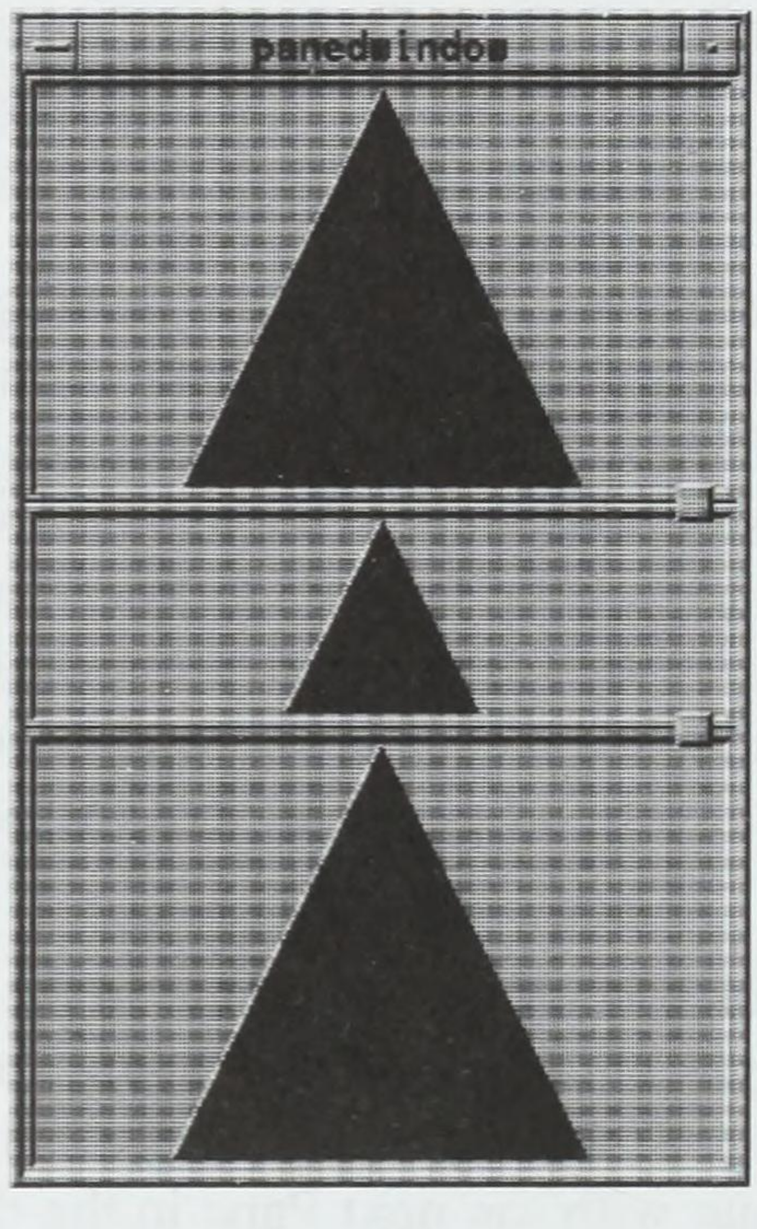
PanedWindow

Description

A PanedWindow should be used to group components into Panes separated by Sashes and Separators for adjusting the relative size of each Pane. This component must be composed of any number of * groups of components, called Panes, each separated by a Sash and a * Separator. The Panes, Sashes, and Separators must be grouped * linearly, either horizontally or vertically. A Sash must be composed * of a handle on the Separator between two Panes for adjusting the * position of the Separator, and therefore the size of the Panes next to * it.

As a Sash is moved, the Pane in the direction of the Sash movement * must get smaller and the other Pane must get larger by an equal * amount. If a Pane is adjusted to its minimum size, adjustment must * continue with the next Pane in the direction of the Sash movement.

Illustration



Navigation

This component must follow the navigation model described in * Chapter 2.

Related Information

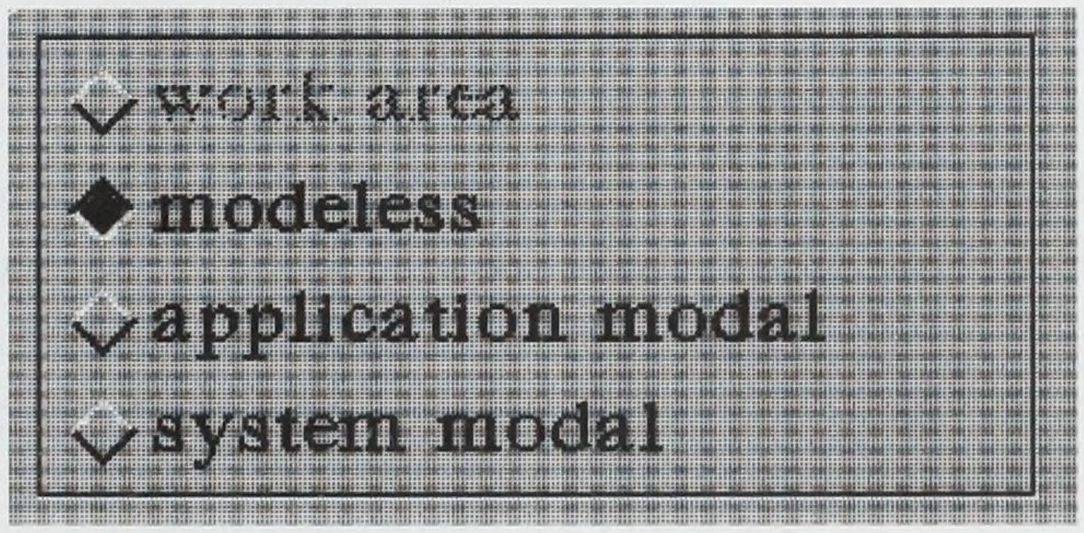
For information about activating a Sash, see the reference page for Sash.

Panel

Description

A Panel group should be used to organize a collection of basic controls in a horizontal, vertical, or 2-dimensional layout. This component is usually composed of just one type of basic control. This component is composed of an area for organizing basic controls.

Illustration



Navigation

KDown, **KLeft**, **KRight**, and **KUp** must navigate among * components in the Panel according to the navigation model * described in Chapter 2.

Common Bindings

Virtual Keys	Default Bindings
KDown	<↓>
KLeft	<←>
KRight	<→>
KUp	<↑>

Pointer Shapes

Description

You should not create new mouse pointer shapes for functions that already have mouse pointer shapes associated with them, except for localization; however, you can create new mouse pointer shapes for functions not already associated with a pointer shape. You must not use a predefined shape to symbolize a function it was not designed to represent.

A list of the defined pointer shapes follows, along with their usage and hotspots. The hotspot of a pointer is the actual position on the pointer that tracks the movements of the mouse. As the pointer changes from one shape to another, the location of the hotspot must not move on the screen. The hotspot is the precise location on the pointer where mouse actions occur.

The upper-left pointing arrow pointer is a general-purpose pointer. It is used in most window areas for single-object selection and activation. The hotspot for the arrow pointer should be in the point of the arrow.

You can optionally use an I-beam pointer in any Text component. It is used to change the location of the text insertion cursor and to perform actions on text. If the I-beam is used, it can be hidden during the time between any keyboard action and a mouse movement. This helps the user distinguish the I-beam pointer from the text insertion cursor, which can also be an I-beam. The hotspot for the I-beam pointer should be on the vertical bar of the I-beam about one-third up from the bottom.

The X pointer can indicate when the pointer is outside of any application area. The hotspot for the X pointer should be where the lines intersect.

The resize pointers indicate positions for area resize, and they remain during a resize operation. The direction of the arrow in the pointer indicates the direction of increasing size. The horizontal and vertical pointers indicate resize in either the horizontal or vertical direction. The diagonal pointers indicate resize in both the horizontal and vertical directions simultaneously. The hotspot for the resizing pointers should be on the elbow or the line at the position pointed to by the arrow.

The hourglass pointer, a working pointer, indicates that an action is in progress in the area, and that the pointer has no effect in the area. While the hourglass pointer is active, all mouse button and keyboard events are ignored in the area. The hotspot for the hourglass pointer should be located at the center of the hourglass, though it should not be used for activation. The hourglass pointer can be used interchangeably with the watch pointer.

The watch pointer, a working pointer, indicates that an action is in progress in the area, and that the pointer has no effect in the area. While the watch pointer is active, all mouse button and keyboard events are ignored in the area. The hotspot for the watch pointer should be located at the top of the watch, though it should not be used for activation. The watch pointer can be used interchangeably with the hourglass pointer.

The 4-directional arrow pointer indicates a move operation in progress, or a resize operation before the resize direction has been determined. During a move operation, the object, or an outline of the object should move to track the location of the pointer. During a resize operation, the pointer is used to indicate a direction for resizing. The 4-directional arrow pointer should change to the appropriate resize arrow when the resize direction is determined, either by crossing an object boundary with the pointer or by pressing a keyboard direction key. The hotspot for the 4-directional arrow pointer should be at the spot where the arrows intersect.

The sighting pointer is used to make fine position selections. For example, in a drawing program it may be used to indicate a pixel to fill or the connecting points of lines. The hotspot for the sighting pointer should be at the spot where the lines intersect.

The caution pointer is used to indicate that action is expected in another area before input can be given to the current area, and that the pointer has no effect in the area. While the caution pointer is active, all mouse button and keyboard events are ignored in the area. The hotspot for the caution pointer should be located at the center of the caution symbol, though it should not be used for activation.

The question pointer is used to request an input position or component from the user. This is often used to input an object for interactive help. The user requests interactive help, then the question pointer is displayed to allow the user to indicate what position or component help is requested for. The hotspot for the question pointer should be at the bottom of the question mark.

You can optionally use an arrow pointing to the upper right corner to indicate a pending Menu action. This shape indicates that a Menu is popped up or pulled down and waiting for a Menu item to be activated or the Menu to be removed. The hotspot for this arrow pointer should be in the point of the arrow.

Related Information

See Chapter 2 for information on the input device model and designing new pointer shapes.

Primary Selection

Description

Each collection maintains its own selection. However, there must only be a single primary selection. The primary selection is the selection on which selection actions, such as cut and copy, are performed. There are two models for how the primary selection moves among collections:

- Focus-based primary selection, where the primary selection must move to a collection whenever the collection gets the input focus.
- Selection-based primary selection, where the primary selection must move to an editable collection whenever a nonnull selection is explicitly started in that collection.

The following primary selection actions, known as the primary transfer actions, should be made available.

BPrimaryPaste Click

Must paste the primary selection to the position of the pointer and set the location cursor there. Whether the paste action cuts or copies the primary selection to the new location depends on the type of elements pasted, and the source and destination of the primary selection as described in Section 3.2.2.

BPrimaryCopy Click

Must copy the primary selection to the position of the pointer and set the location cursor there.

BPrimaryCut Click

Must cut the primary selection to the position of the pointer and set the location cursor there.

KPrimaryPaste

Must paste the primary selection to the position of the location cursor. Whether the paste action cuts or copies

the primary selection to the new location depends on the type of elements pasted, and the source and destination of the primary selection as described in Section 3.2.2.

KPrimaryCopy

Must copy the primary selection to the position of the location cursor. *

KPrimaryCut

Must cut the primary selection to the position of the location cursor. *

KReselect

Can optionally restore the collection's selection as the primary selection.

Common Bindings

Virtual Key	Default Bindings
BPrimaryPaste	BDrag
BPrimaryCopy	MCtrl BPrimaryPaste
BPrimaryCut	MAlt BPrimaryPaste
KPrimaryPaste	<vendor defined>
KPrimaryCopy	MCtrl KPrimaryPaste
	MAlt KCopy
KPrimaryCut	MAlt KPrimaryPaste
	MAlt KCut
KReselect	MCtrl MShift KSelect

Related Information

See Section 3.2.2 for more information about the primary selection.

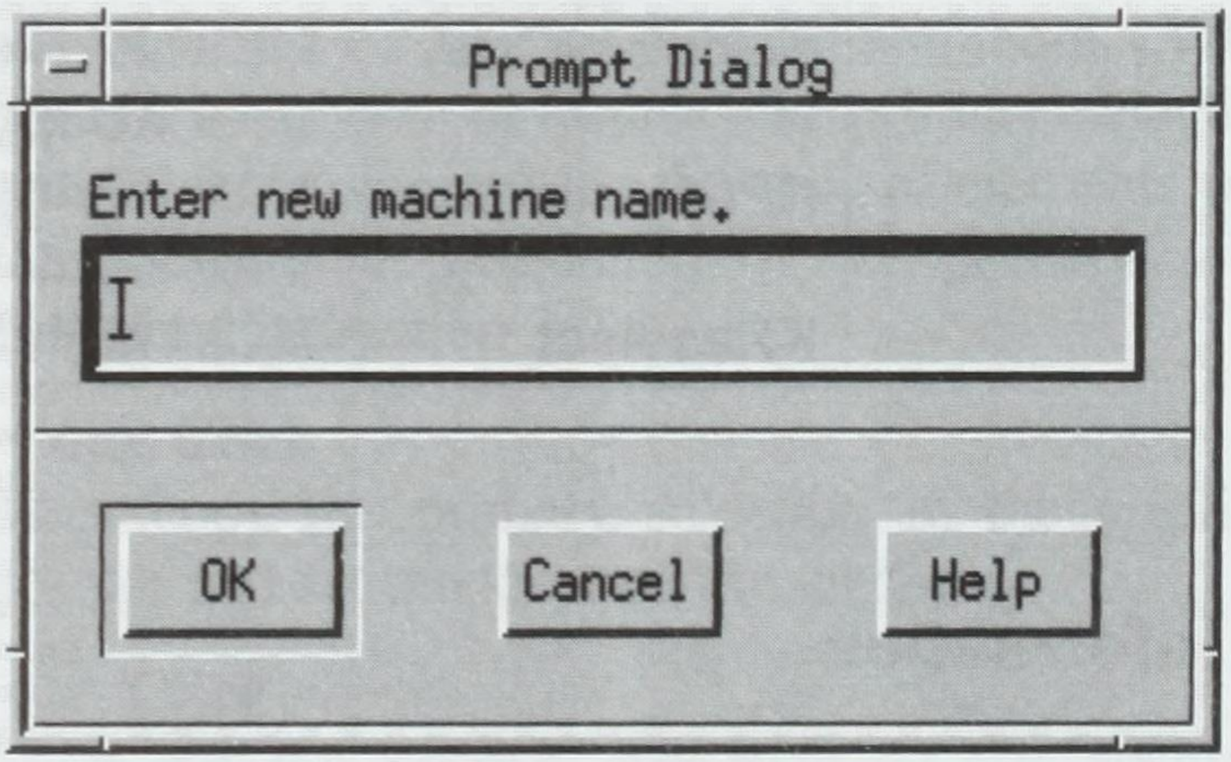
PromptDialog

Description

A PrompDialog should be used to prompt the user for input. It can interrupt the user's interaction with the application. It should include a message, a text input area, and one of the following button arrangements.

OK Cancel
OK Cancel Help
OK Apply Cancel
OK Apply Cancel Help
OK Apply Reset Cancel
OK Apply Reset Cancel Help

Illustration



Related Information

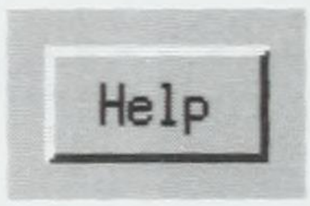
For more information, see the reference page for DialogBox.

PushButton

Description

A PushButton should be used to start an operation. A PushButton must contain either a text or graphic label indicating the operation of the button. *

Illustration



Navigation

A PushButton must have no internal navigation. *

Other Operations

The following text describes the operations of this component.

BSelect Press

Must arm the PushButton. *

BSelect Release

If the release is within the same PushButton as the press, **BSelect Release** must disarm the PushButton and * *

activate it. If the release is outside of the PushButton, **BSelect Release** must disarm the PushButton without activating it. *

KActivate Must activate the PushButton. *

KSelect Must activate the PushButton. *

KHelp Must provide any available help for the PushButton. *

Common Bindings

Virtual Keys	Default Bindings
BSelect	<Btn1>
KEnter	<Return>
	<Enter>
KActivate	KEnter
	MCtrl KEnter
KSelect	<Select>
	<Space>
	MCtrl <Space>
KHelp	<Help>
	<F1>

QuestionDialog

Description

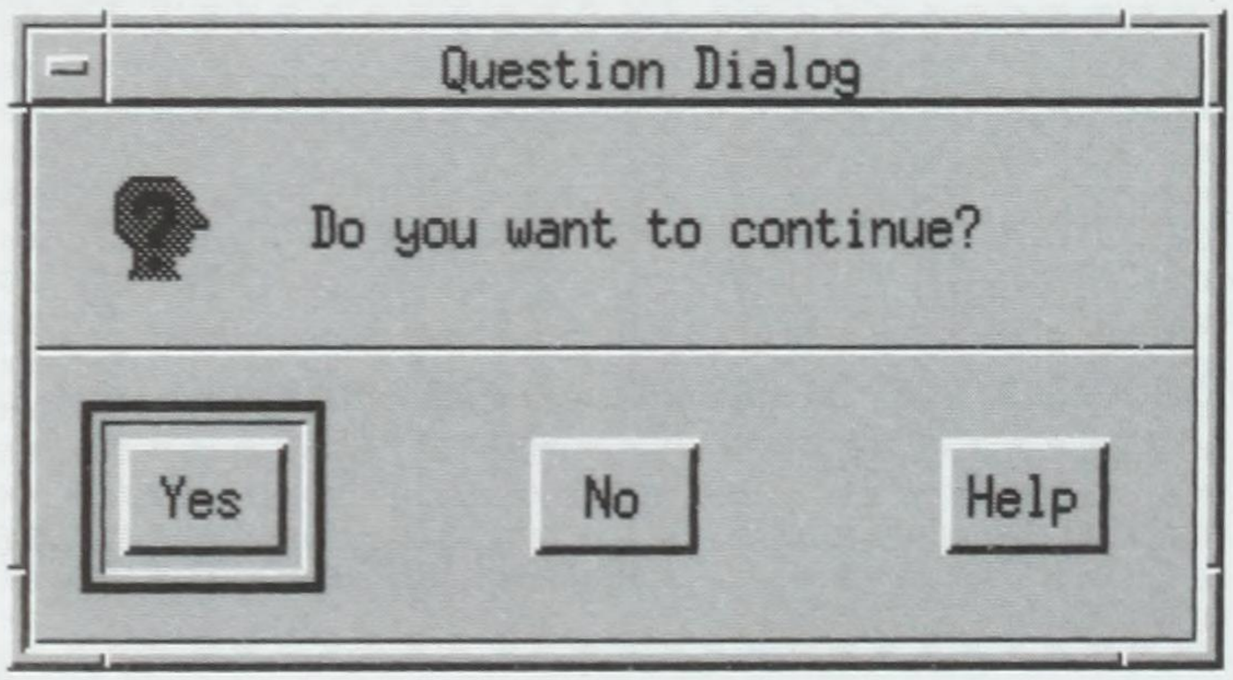
A QuestionDialog should be used to get a user response to a question. It should interrupt the user's interaction with the application. It should include a question symbol, a message, and one of the following button arrangements.

Yes No
Yes No Help

It is possible that both the **Yes** and **No** actions of a Question DialogBox will perform an action. If this is the case, the Question DialogBox should use one of the following button arrangements:

Yes No Cancel
Yes No Cancel Help

Illustration



Related Information

For more information, see the reference page for DialogBox.

Quick Transfer

Description

Quick transfer is used to make a selection, and then immediately cut or copy that selection to the destination cursor. In quick transfer, the user picks the destination first, then makes a selection to be pasted there. This allows multiple selections to be quickly pasted at the same location.

The destination cursor must be in the component in which the user last did a selection, insertion, or editing operation. It can either be at the location cursor of that component, or determined independently. *

If the destination cursor is determined independently, it must be at the location where selection, insertion, or editing was last done. Furthermore, when the current selection is null, the anchor must be kept at the destination cursor as well. The destination cursor should be determined independently in environments in which strong support of keyboard use is expected. * * *

The following quick-transfer actions should be made available.

BQuickPaste Drag Must select a region for quick transfer, and paste the region at the destination cursor when the button is released. Whether the paste action cuts or copies the selection to the new location depends on the type of elements pasted, and the source and destination of the selection as in primary selection described in Section 3.2.2. * *

BQuickCopy Drag Must copy the selection to the position of the pointer. *

BQuickCut Drag Must cut the selection to the position of the pointer. *

KQuickPaste	Must set a quick anchor point to start a quick-paste operation. Whether the paste action cuts or copies the selection to the new location depends on the type of elements pasted, and the source and destination of the selection as in primary selection described in Section 3.2.2.	*
KQuickCopy	Must set a quick anchor point to start a quick-copy operation.	*
KQuickCut	Must set a quick anchor point to start a quick-cut operation.	*
KQuickExtend	Must perform the quick operation, transferring the elements between the quick anchor point and the location cursor.	* *

Common Bindings

Virtual Key	Default Bindings
BQuickPaste	BDrag
BQuickCopy	MCtrl BQuickPaste
BQuickCut	MAlt BQuickPaste
KQuickPaste	<vendor defined>
KQuickCopy	MCtrl KQuickPaste
KQuickCut	MAlt KQuickPaste
KQuickExtend	MShift KQuickPaste

Related Information

See Section 3.2.3 for more information about quick transfer.

RadioButton

Description

A RadioButton should be used to select one option from a number of options. A RadioButton is a special case of a ToggleButton. Only one RadioButton can be set at a time. This component must * be composed of a text or graphic label, and a graphic that indicates * the state of the RadioButton. The graphic should be a diamond preceding the label, and should have two distinctive states that indicate the set and unset states of the RadioButton. On color systems, the on state color can be distinct from general application colors to visually distinguish the on state.

Illustration

Navigation

RadioButtons must have no internal navigation. *

Other Operations

The following text describes the operations of this component.

BSelect Press

Must arm the RadioButton. If the RadioButton was previously unset, it must show the RadioButton in the set state. * * *

BSelect Release

If the release happens in the same RadioButton that the press occurred in, then if the RadioButton was previously unset, it must be set, and any other RadioButton in the same Panel that was previously set must be unset. The RadioButton must be disarmed, and if the RadioButton was in a Menu, the Menu must be unposted. * * * * * * * *

KActivate If the RadioButton was previously unset, it must be set, and any other RadioButton in the same Panel that was previously set must be unset. The RadioButton must be disarmed, and if the RadioButton was in a Menu, the Menu must be unposted. * * * * * *

KSelect If the RadioButton was previously unset, it must be set, and any other RadioButton in the same Panel that was previously set must be unset. The RadioButton must be disarmed, and if the RadioButton was in a Menu, the Menu must be unposted. * * * * * *

KHelp Must provide any available help for the RadioButton. *

Common Bindings

Virtual Keys	Default Bindings
BSelect	<Btn1>
KEnter	<Return>
KActivate	<Enter>
	MCtrl <Enter>
KSelect	<Select>
	<Space>
	MCtrl <Space>
KHelp	<Help>
	<F1>

Related Information

For more information, see the reference pages for CheckButton and ToggleButton.

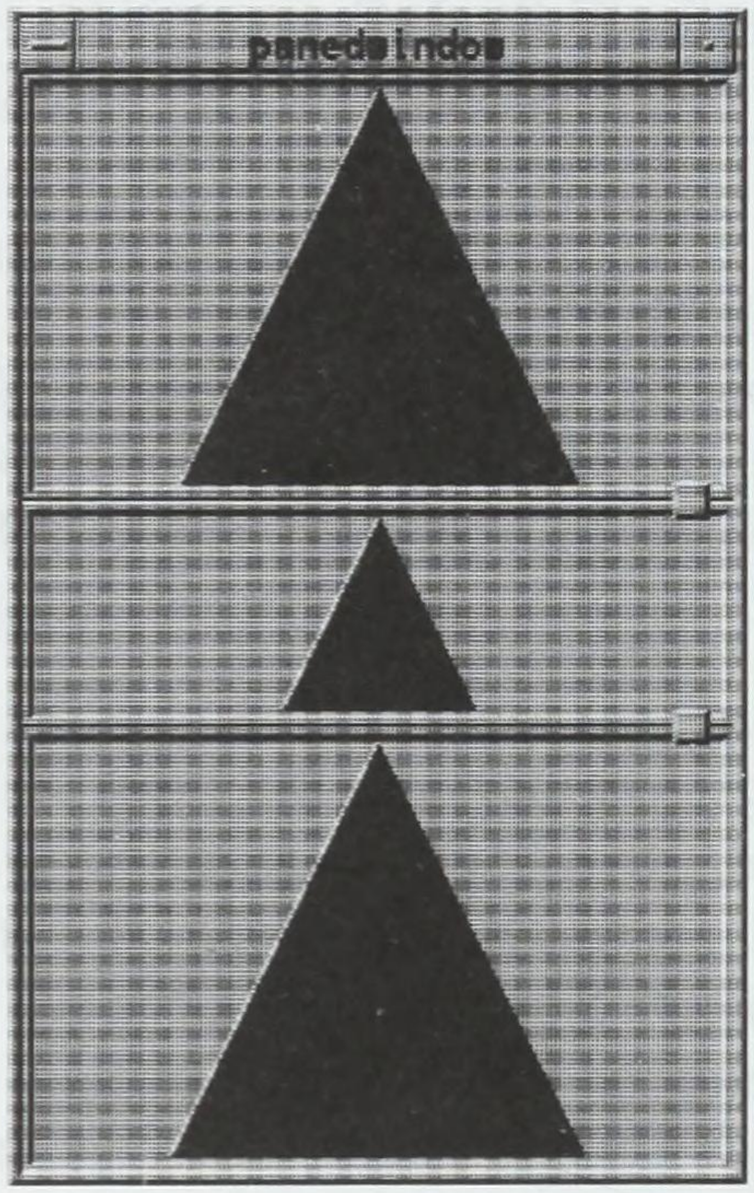
Sash

Description

A Sash should be used to adjust the border between groups of components in a PanedWindow. A Sash must be composed of a handle on the Separator between two Panes for adjusting the position of the Separator, and therefore the size of the Panes next to it.

As a Sash is moved, the Pane in the direction of the Sash movement must get smaller and the other Pane must get larger by an equal amount. If a Pane is adjusted to its minimum size, adjustment should continue with the next Pane in the direction of the Sash movement. The PanedWindow can optionally support movement of the Pane in the initial direction of mouse movement with the modifier **MShift** on button events, rather than resizing the Pane.

Illustration



Navigation

A Sash must have no internal navigation. *

Other Operations

The following text describes the operations of this component.

BSelect or **BDrag Drag**
Must cause the Sash to track the movement of the pointer. In a vertically oriented PanedWindow, the Sash must track the vertical position of the pointer. In a horizontally oriented PanedWindow, the Sash must track the horizontal position of the pointer. *

BSelect or **BDrag Drag**
Can cause the Pane in the initial direction of movement to track the movement of the pointer. In a vertically oriented PanedWindow, the Pane must track the vertical position of the pointer. In a horizontally oriented PanedWindow, the Pane must track the horizontal position of the pointer. *

KUp, **KDown**, **KRight**, or **KLeft**
Must cause the Sash to move in the specified direction one increment. *

MShift KUp, **KDown**, **KRight**, or **KLeft**
Can cause the Pane in the initial direction to move one increment in the specified direction.

MCtrl KUp, **KDown**, **KRight**, or **KLeft**
Must cause the Sash to move in the specified direction one large increment. *

MCtrl MShift KUp, **KDown**, **KRight**, or **KLeft**
Can cause the Pane in the initial direction to move one large increment in the specified direction.

KHelp Must provide any available help for the Sash. *

Common Bindings

Virtual Keys	Default Bindings
BSelect	<Btn1>
BDrag	<Btn2>
KDown	<↓>
KLeft	<←>
KRight	<→>
KUp	<↑>
KHelp	<Help>
	<F1>

Related Information

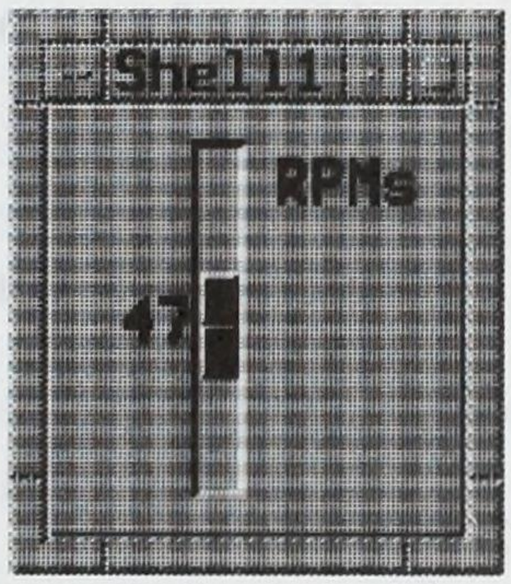
For more information, see the reference page for PanedWindow.

Scale

Description

A Scale should be used to select a value from a range. This component should be composed of a slider, moving within an element that indicates the size of the range, called the trough, and a label that indicates the current value. The position of the slider indicates the value relative to the range. A Scale can also have buttons with arrow graphics for moving the slider.

Illustration



Other Operations

The following text describes the operations of this component.

BSelect Press

In an arrow button, this action must move the slider one increment in the direction of the side of the slider on which the button was pressed, and autorepeat until the button is released.

In the trough, if the Scale has tick marks, **BSelect Press** must move the slider one tick mark in the direction of the side of the slider on which the trough was pressed, and autorepeat until the button is released. It can stop when the slider reaches the position of the pointer. If the Scale does not have tick marks, **BSelect Press** in the trough must move the slider one large increment in the direction of the side of the slider on which the trough was pressed, and autorepeat until the button is released. It can stop when the slider reaches the position of the pointer.

BSelect Drag

Must cause the slider to track the position of the pointer. In a vertical Scale, the slider must track the vertical position of the pointer. In a horizontal Scale, the slider must track the horizontal position of the pointer.

MCtrl BSelect Press

In the slider trough or an arrow button, this action must move the slider to the end of the Scale on which the button was pressed.

BDrag Press

In the trough, this action can directly position the slider to the position of the button press.

BDrag Drag

Must cause the slider to track the position of the pointer. In a vertical ScrollBar, the slider must track the vertical position of the pointer. In a horizontal

ScrollBar, the slider must track the horizontal position of the pointer. *

In the trough, this action can directly position the slider to the position of the button press, and then cause the slider to track the position of the pointer.

KDown For vertical Scales, this action must move the slider one increment down. *

KUp For vertical Scales, this action must move the slider one increment up. *

KLeft For horizontal Scales, this action must move the slider one increment left. *

KRight For horizontal Scales, this action must move the slider one increment right. *

MCtrl KDown
For vertical Scales, this action must move the slider one large increment down. *

MCtrl KUp
For vertical Scales, this action must move the slider one large increment up. *

MCtrl KLeft
For horizontal Scales, this action must move the slider one large increment left. *

MCtrl KRight
For horizontal Scales, this action must move the slider one large increment right. *

KBeginLine or **KBeginData**
Must move the slider to the minimum value. *

KEndLine or **KEndData**
Must move the slider to the maximum value. *

KHelp Must provide any available help for the Scale. *

Common Bindings

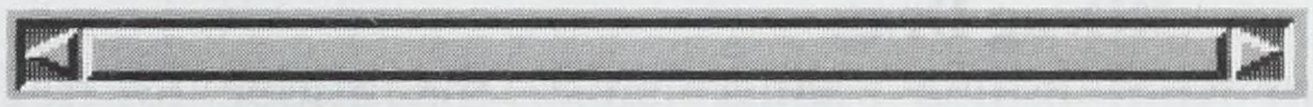
Virtual Keys	Default Bindings
BSelect	<Btn1>
BDrag	<Btn2>
KDown	<↓>
KLeft	<←>
KRight	<→>
KUp	<↑>
KBeginLine	<Begin>
	<Home>
KBeginData	MCtrl <Begin>
	MCtrl <Home>
KEndLine	<End>
KEndData	MCtrl <End>
KHelp	<Help>
	<F1>

ScrollBar

Description

A ScrollBar should be used to scroll the visible area of another component. A ScrollBar should be composed of a slider, moving within an element that indicates the full size of the scrolled component, and two buttons with arrow graphics for moving the slider. The slider indicates the relative position and size of the visible area of the scrolled component.

Illustration



Other Operations

The following text describes the operations of this component.

BSelect Press

In an arrow button, this action must move the slider one increment in the direction of the side of the slider on which the button was pressed, and autorepeat until the button is released. *

In the trough, this action must move the slider one page in the direction of the side of the slider on which the trough was pressed, and autorepeat until the button is released. It can stop when the slider reaches the position of the pointer. *

BSelect Drag

Must cause the slider to track the position of the pointer. In a vertical ScrollBar, the slider must track the vertical position of the pointer. In a horizontal ScrollBar, the slider must track the horizontal position of the pointer. *

MCtrl BSelect Press

In the slider trough or an arrow button, this action must move the slider to the end of the ScrollBar on which the button was pressed. *

MShift BSelect Press

Can scroll the view so that the cursor within that view is positioned at the edge of the view region on the side of the ScrollBar on which the button was pressed.

BDrag Press

In the trough, this action can directly position the slider to the position of the button press.

BDrag Drag

Must cause the slider to track the position of the pointer. In a vertical ScrollBar, the slider must track the vertical position of the pointer. In a horizontal ScrollBar, the slider must track the horizontal position of the pointer. *

In the trough, this action can directly position the slider to the position of the button press, and then cause the slider to track the position of the pointer.

KDown For vertical ScrollBars, this action must move the slider one increment down. *

KUp For vertical ScrollBars, this action must move the slider one increment up. *

KLeft For horizontal ScrollBars, this action must move the slider one increment left. *

KRight For horizontal ScrollBars, this action must move the slider one increment right. *

MCtrl KDown
For vertical ScrollBars, this action must move the slider one large increment down. *

MCtrl KUp
For vertical ScrollBars, this action must move the slider one large increment up. *

MCtrl KLeft
For horizontal ScrollBars, this action must move the slider one large increment left. *

MCtrl KRight
For horizontal ScrollBars, this action must move the slider one large increment right. *

KPageDown
For vertical ScrollBars, this action must move the slider one page down. *

KPageUp For vertical ScrollBars, this action must move the slider one page up. *

KPageLeft For horizontal ScrollBars, this action must move the slider one page left. *

KPageRight
For horizontal ScrollBars, this action must move the slider one page right. *

KBeginLine or **KBeginData**
Must move the slider to the minimum value. *

KEndLine or **KEndData**
Must move the slider to the maximum value. *

KHelp Must provide any available help for the ScrollBar. *

Common Bindings

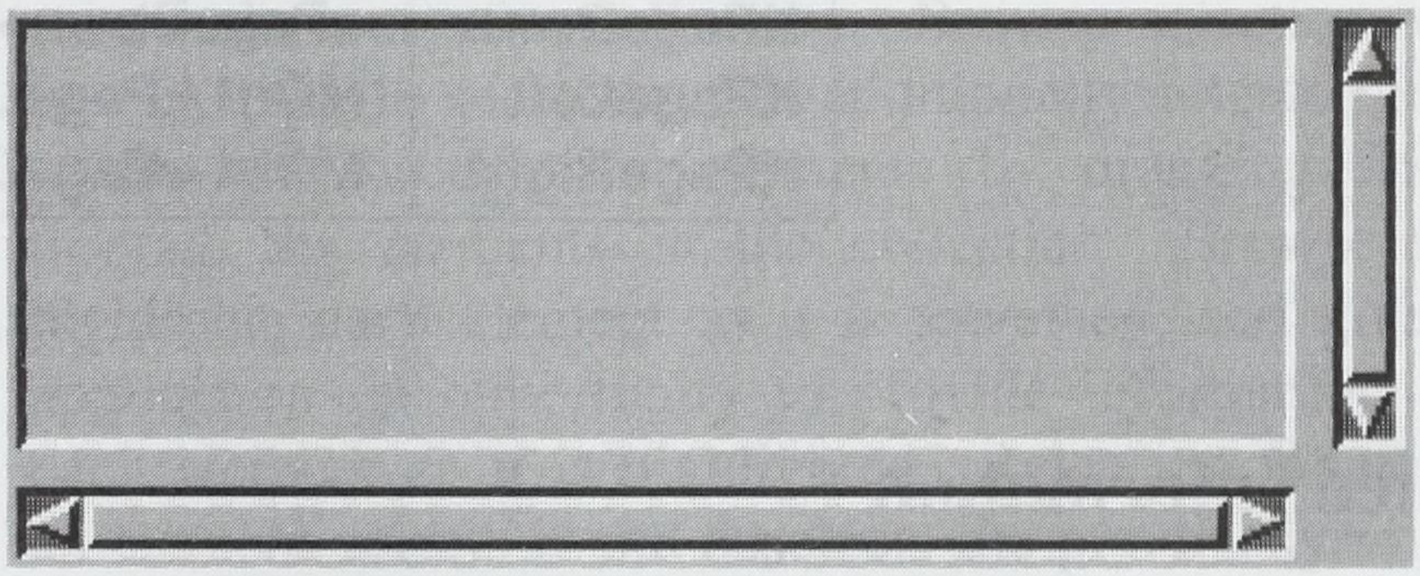
Virtual Keys	Default Bindings
BSelect	<Btn1>
BDrag	<Btn2>
KDown	<↓>
KLeft	<←>
KRight	<→>
KUp	<↑>
KPageUp	<PageUp>
KPageDown	<PageDown>
KPageLeft	MCtrl <PageUp>
KPageRight	MCtrl <PageDown>
KBeginLine	<Begin>
	<Home>
KBeginData	MCtrl <Begin>
	MCtrl <Home>
KEndLine	<End>
KEndData	MCtrl <End>
KHelp	<Help>
	<F1>

ScrolledWindow

Description

A ScrolledWindow should be used to frame other components and to provide ScrollBars as necessary to scroll the visible area of the framed components. When the area framed by the ScrolledWindow is completely displayed, the ScrollBars need not be displayed. * When the area provided in the ScrolledWindow is too small to * display the entire component area, the ScrollBars must be displayed * and must then allow scrolling of the visible area.

Illustration



Navigation

A ScrolledWindow must follow the navigation model described in * Chapter 2.

Other Operations

The following text describes the operations of this component.

KPageDown
Must scroll the window down one page. *

KPageLeft
Must scroll the window left one page. *

KPageRight
Must scroll the window right one page. *

KPageUp Must scroll the window up one page. *

Common Bindings

Virtual Keys	Default Bindings
KPageUp	<PageUp>
KPageDown	<PageDown>
KPageLeft	MCtrl <PageUp>
KPageRight	MCtrl <PageDown>

Related Information

For information about the activation of the ScrollBars, see the reference page for ScrollBar. For more information about navigation within general scrollable components, see Section 2.3.2.4.

Selection

Description

The selection model determines how elements are selected from a group of elements. OSF/Motif compliant systems must support the five different selection models. The five selection models are as follows: *

Single Selection *

Used to select a single element in a collection. Clicking **BSelect** on an element selects it and deselects the previously selected element in the collection. Single selection is described in Section 3.1.1 *

Browse Selection *

Used to allow browsing through single selection collections. Browse selection is also used to select a single element of a collection. Browse selection works just like single selection, but additionally allows the user to browse through the elements by dragging **BSelect** through the elements in the collection. Browse select highlights each element as it is traversed, and gives the application an opportunity to provide information about each element as it is highlighted. Releasing **BSelect** on an element selects it and deselects the previously selected element. Browse selection is described in Section 3.1.2 *

Multiple Selection *

Used to select or deselect multiple elements of a collection. Clicking **BSelect** on an unselected element adds that element to the current selection. Clicking **BSelect** on a selected element removes that element from the current selection. Multiple selection is described in Section 3.1.3 *

Range Selection *

Used to select a contiguous range of elements in a collection. Clicking **BSelect** on an element selects the

single element, and deselects any previous selection. **BSelect Drag** over a range of elements selects all the elements within the range, and deselects any previous selection. Range selection is described in Section 3.1.4 *

Discontiguous Selection *

Used for simultaneously selecting multiple discontiguous ranges of elements in a collection. Clicking or dragging **BSelect** operates just the same as for range selection. Discontiguous selection also allows **BToggle** to be used to add new selection ranges to or remove selection ranges from the selection. Discontiguous selection is described in Section 3.1.5.

Related Information

See Chapter 3 for more information about the selection models.

SelectionBox

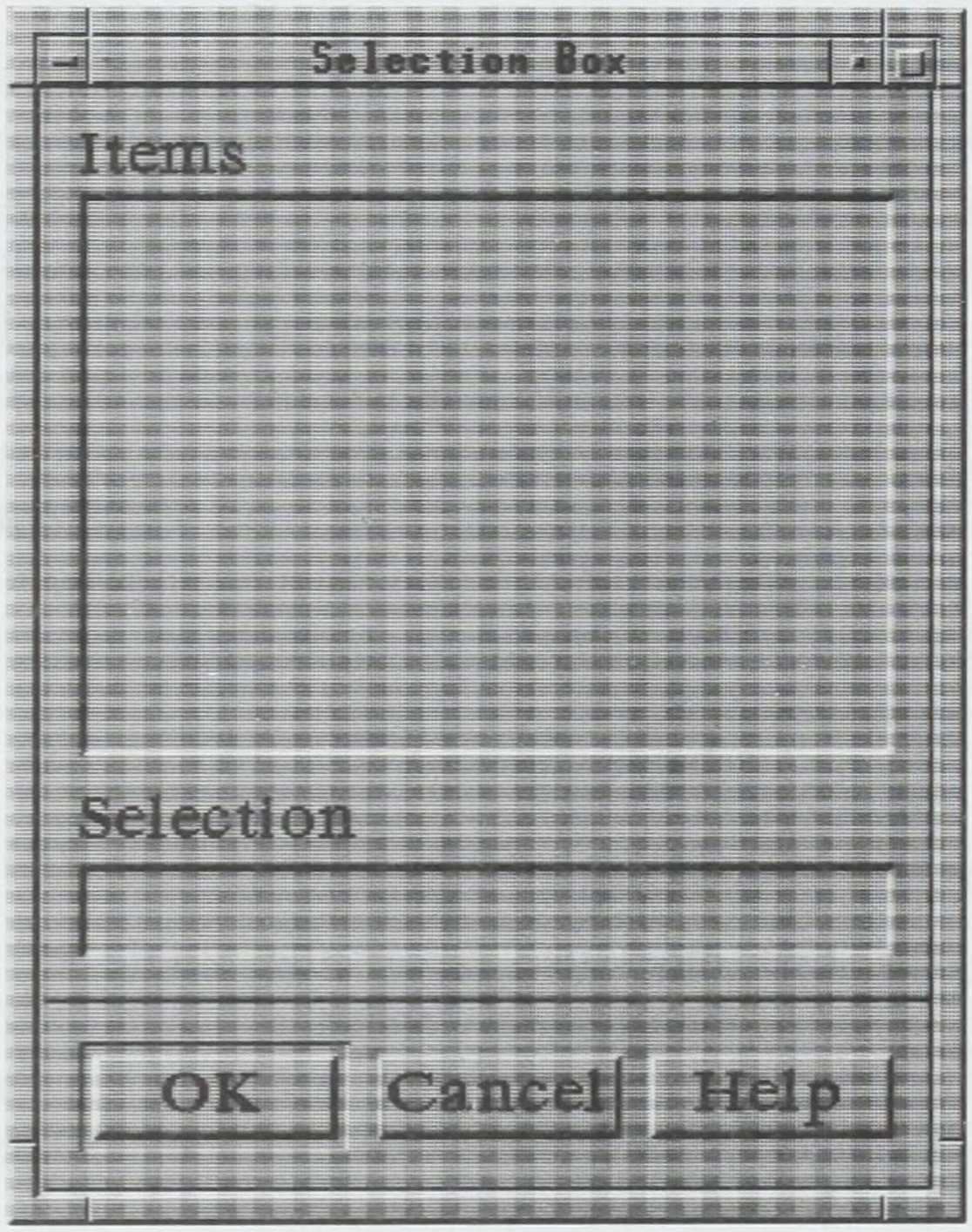
Description

A SelectionBox is a special purpose composite component for making selections from a list of choices. The SelectionBox must be composed of a Text component for the selected alternative, and a List component above the Text component for presenting the alternatives. Both the List and Text components should have a Label above them. The List component can be scrollable. The SelectionBox should also include one of the following button arrangements.

OK Cancel Help
OK Cancel Apply Help

The List must use either the single or browse selection model. Selecting an element from the list must place the selected element in the Text component. Entering an item name into the Text component must make that element visible in the List component.

Illustration



Related Information

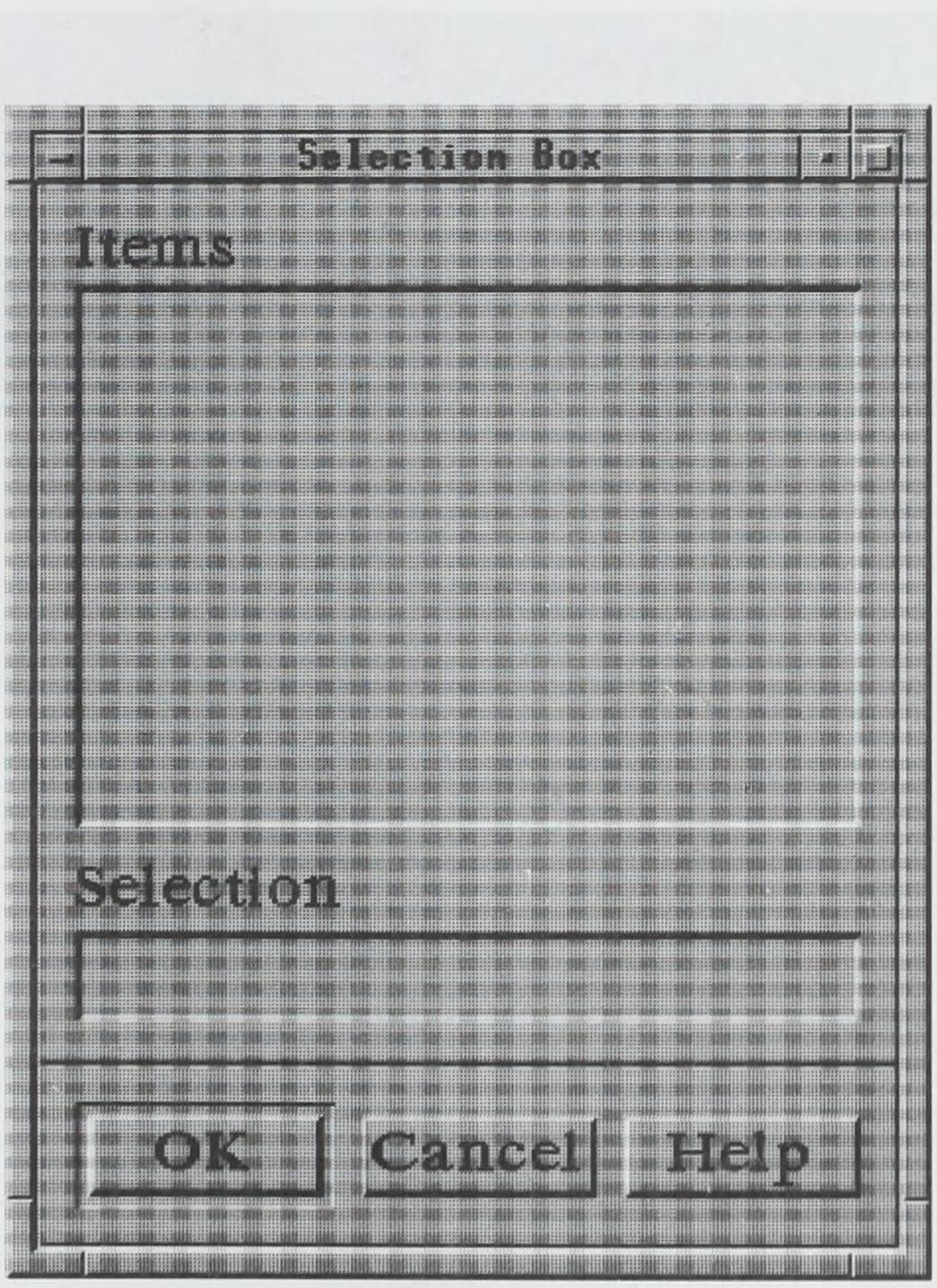
For more information, see the reference page for Selection DialogBox.

SelectionDialog

Description

A SelectionDialog should be used to allow a user to make a selection from a list of choices. It can interrupt the user's interaction with the application. It should contain a SelectionBox.

Illustration



Related Information

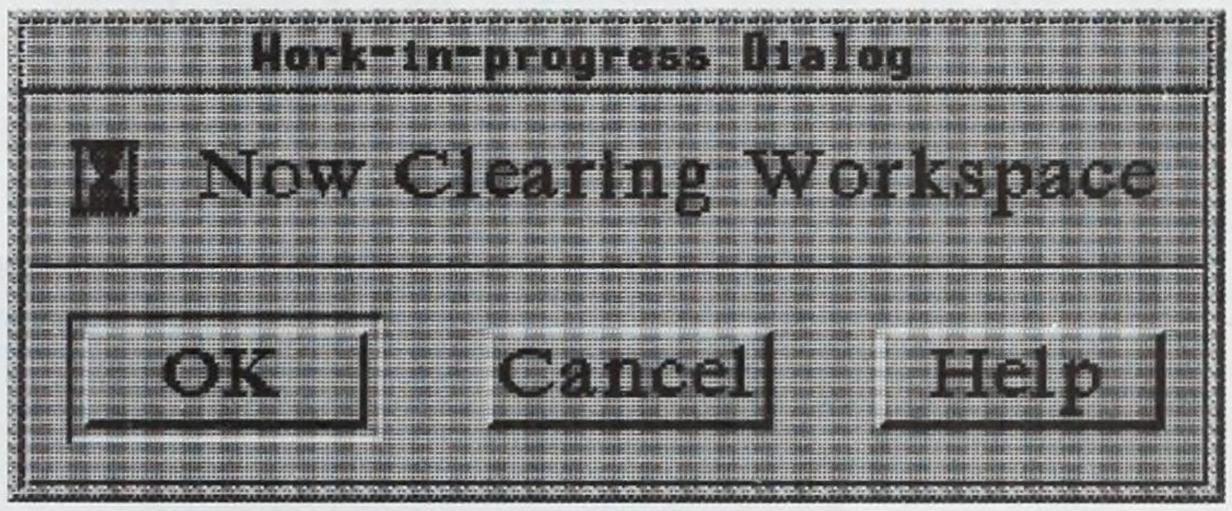
For more information, see the reference page for DialogBox.

Separator

Description

A Separator should be used to separate elements of the application. It should be composed of a vertical or horizontal line.

Illustration



Navigation

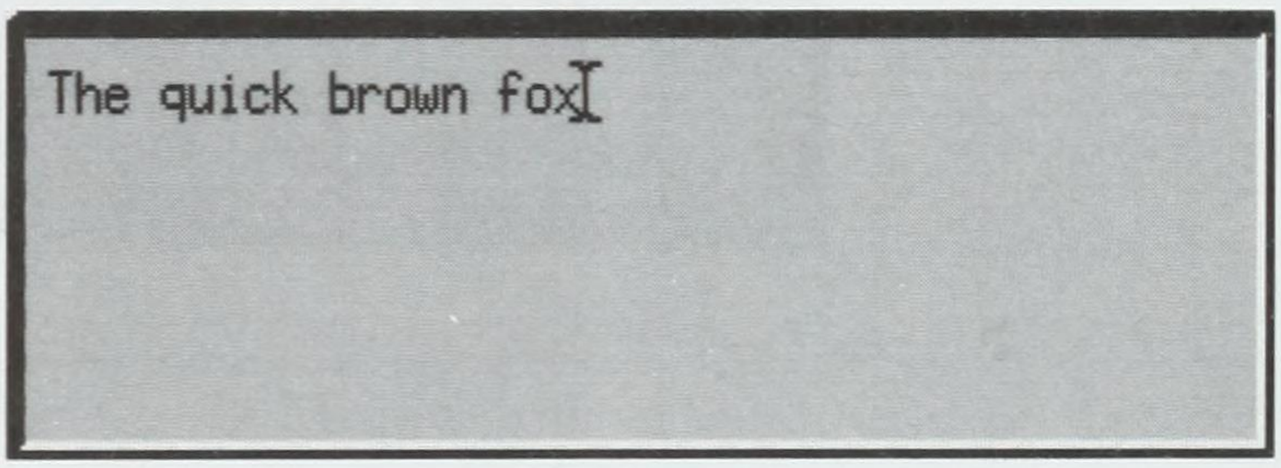
A Separator must have no internal navigation. *

Text

Description

A Text component should be used to display and enter text. A Text component must be composed of an area for displaying and entering text. The text can be either a single line or multiple lines. Text must support the range selection model as described in Chapter 3 and can support the discontiguous selection model.

Illustration



Navigation

The following text describes the navigation actions of this component.

KUp In multiline Text components, **KUp** must move the location cursor up one line. In single-line Text components, **KUp** must navigate upwards to the next component if the Text is designed to act like a basic control.

KDown In multiline Text components, **KDown** must move the location cursor down one line. In single-line Text

components, **KDown** must navigate downwards to the next component if the Text is designed to act like a basic control. *

KLeft Must move the location cursor left one character. *

KRight Must move the location cursor right one character. *

KNextWord
Must move the location cursor to the right by a word; that is, the first character after a space, tab, or newline character. In single-line Text controls, **KNextWord** can navigate rightward to the next component if the Text is designed to act like a basic control. * *

KPrevWord
Must move the location cursor to the left by a word; that is, the first nonblank character after the first previous blank character. In single-line Text controls, **KPrevWord** can navigate leftward to the next component if the Text is designed to act like a basic control. * *

KNextPara
In multiline Text components, **KNextPara** can optionally move the location cursor to the beginning of the next paragraph.

KPrevPara
In multiline Text components, **KPrevPara** can optionally move the location cursor to the beginning of the previous paragraph.

KBeginLine
Must move the location cursor to the beginning of the line. *

KEndLine Must move the location cursor to the end of the line. *

KBeginData
In multiline Text components, **KBeginData** must move the location cursor to the beginning of the file. *

KEndData

In multiline Text components, **KEndData** must move * the location cursor to the end of the file.

Other Operations

The following text describes the operations of this component.

KPageUp In multiline Text components, **KPageUp** must scroll the * viewing window up one page of text, displaying the * previous page of text.

KPageDown

In multiline Text components, **KPageDown** must scroll * the viewing window down one page of text, displaying * the next page of text.

KPageLeft

Must scroll the viewing window left one page of text. *

KPageRight

Must scroll the viewing window right one page of text. *

KSpace Must insert a space. Modifying this, or any **KSelect** * binding, with **MCtrl** must evoke its normal selection * function.

KEnter In multiline Text components, **KEnter** must insert a * carriage return. Modifying this, or any **KActivate** * binding, with **MCtrl** must evoke its normal activation * function. When the location cursor is at the end of the text, **KEnter** can optionally traverse to the next field.

KTab and **KBackTab**

In single-line Text components, these must have no effect except that **KTab** can optionally traverse to the next field. In multiline Text components, these can optionally move the location cursor to the next or previous tab stop. Another option is for **KTab** to insert a tab character, and for **KBackTab** to have no effect. *

Modifying **Tab** or **BackTab** with **MCtrl** must evoke their normal field navigation functions. When the location cursor is at the end of the text, **KTab** can optionally traverse to the next field. *

KBackSpace

If any characters are selected and the cursor is in or at the bounds of the selection, **KBackSpace** must delete the selected characters; otherwise, it must delete the character preceding the location cursor. *

KDelete If any characters are selected and the cursor is in or at the bounds of the selection, **KDelete** must delete the selected characters; otherwise, it must delete the character following the location cursor. *

KClear Can optionally delete the selected text, and replace it with blank space.

KEraseEndLine

Can optionally delete the characters following the location cursor to the end of the line.

KInsert Can optionally toggle between insert mode and replace mode. Text components must start in insert mode, where the location cursor is between two characters. In insert mode, typing a character must insert the character at the position of the location cursor. Text components can also implement replace mode, where the location cursor is on a character. In replace mode, typing a character must replace the current character with that newly entered character and move the location cursor to the next character. *

BSelect Click 2+

Selects text a block at a time. Double-click must select *

text a word at a time. Triple-click can optionally select text a line or sentence at a time. Larger numbers of clicks can optionally select increasingly larger blocks of text.

BSelect Drag 2+
Can drag out a new selection by blocks as described for **BSelect Click 2+**.

BExtend Click 2+
Can extend the selection to a block boundary as described for **BSelect Click 2+**.

BExtend Drag 2+
Can extend the selection by blocks as described for **BSelect Click 2+**.

BToggle Click 2+
If the Text component supports discontiguous selection, this can extend the discontiguous selection by blocks as described for **BSelect Click 2+**.

BToggle Drag 2+
If the Text component supports discontiguous selection, this can drag out the discontiguous selection by blocks as described for **BSelect Click 2+**.

KHelp Must provide any available help for the Text control. *

Common Bindings

Virtual Keys	Default Bindings
BSelect	<Btn1>
BExtend	MShift BSelect
BToggle	MCtrl BSelect
KUp	<↑>

Virtual Keys	Default Bindings
KDown	<↓>
KLeft	<←>
KRight	<→>
KPageUp	<PageUp>
KPageDown	<PageDown>
KPageLeft	MCtrl <PageUp>
KPageRight	MCtrl <PageDown>
KNextWord	MCtrl KRight
KPrevWord	MCtrl KLeft
KNextPara	MCtrl KDown
KPrevPara	MCtrl KUp
KEndLine	<End>
KEndData	MCtrl <End>
KBeginLine	<Begin>
	<Home>
KBeginData	MCtrl <Begin>
	MCtrl <Home>
KSelect	<Select>
	KSpace
	MCtrl KSpace
KSpace	<space>
KActivate	KEnter
	MCtrl KEnter
KEnter	<Enter>
	<Return>
KTab	<Tab>
KBackTab	<BackTab>
	MCtrl <Tab>
KBackSpace	<BackSpace>
KDelete	<Delete>
KClear	<Clear>
KEraseEndLine	MCtrl KDelete
KInsert	<Insert>
KHelp	<Help>
	<F1>

Related Information

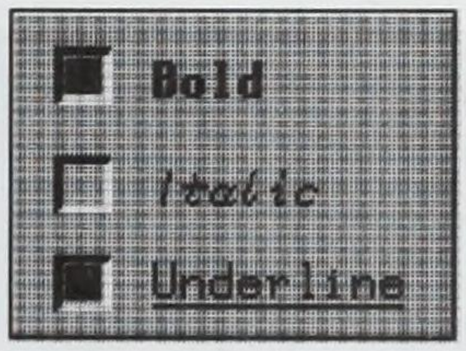
For more information on Text selection, see the description of the selection models in Chapter 3.

ToggleButton

Description

A ToggleButton should be used to select options in an application. When the choice is one of many, the ToggleButton is called a RadioButton. When the choice is any of many, the ToggleButton is called a CheckButton. The ToggleButton must be composed of a * text or graphic label, and a can contain a graphic that indicates the state of the ToggleButton. The graphic should precede the label, and should have two distinctive states that indicate the set and unset states of the ToggleButton.

Illustration



Navigation

ToggleButtons must have no internal navigation. *

Related Information

For a description of the activation of the two types of ToggleButtons, see the reference pages for CheckButton and RadioButton.

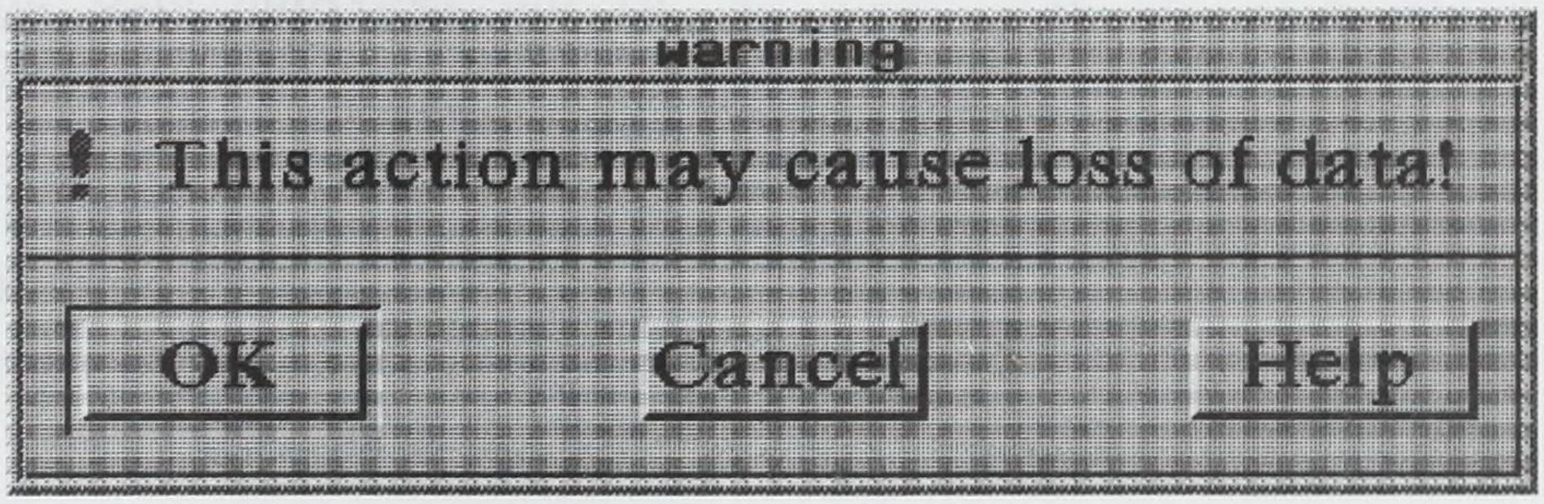
WarningDialog

Description

A WarningDialog should be used to alert the user to a possible danger. It should interrupt the user's interaction with the application. It should contain a warning symbol, a message, and one of the following button arrangements.

Yes No
Yes No Help
OK Cancel
OK Cancel Help

Illustration



Related Information

For more information, see the reference page for DialogBox.

Window Menu

Description

The window Menu, sometimes called the system Menu, is used to display the list of window actions. All actions possible for a window should be displayed in the window Menu, since keyboard-only users interact with the window manager through this Menu.

The window Menu button must be located in the upper-left corner of the window. Double-clicking the window Menu button must close the window. Pressing **KWindowMenu** must activate the window Menu button for the window with the focus. The illustration on this reference page shows a posted window Menu.

A primary window Menu must have the following selections in the order listed:

Restore Alt+F5
Restores a minimized or maximized window to the previous size and location of the window. This selection must be deemphasized (grayed out) when the window is in its normal state. This action must have the mnemonic **R**. If this action has an accelerator, it must be **MAlt <F5>** if **<F5>** is available.

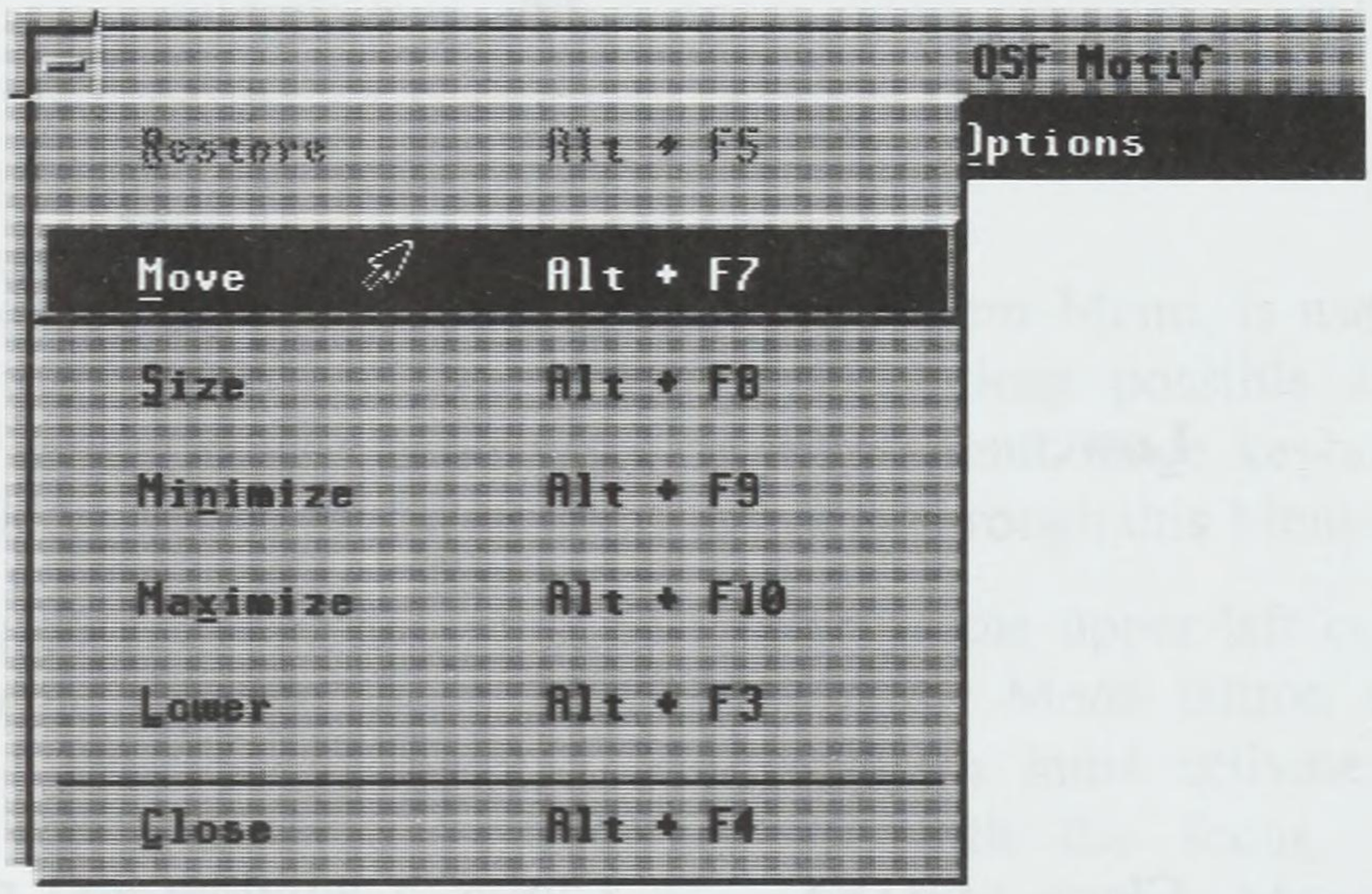
Move Alt+F7
Moves a window around the workspace. This action must have the mnemonic **M**. If this action has an accelerator, it must be **MAlt <F7>** if **<F7>** is available.

Size Alt+F8
Changes the height and width of the window in the direction indicated by the pointer. This action must have the mnemonic **S**. If this action has an accelerator, it must be **MAlt <F8>** if **<F8>** is available.

Minimize	**Alt+F9**	Changes a window into an icon. This action must have the mnemonic **N**. If this action has an accelerator, it must be **MAlt <F9>** if **<F9>** is available.
Maximize	**Alt+F10**	Enlarges a window to its maximum size. This action must have the mnemonic **X**. If this action has an accelerator, it must be **MAlt <F10>** if **<F10>** is available.
Lower	**Alt+F3**	Moves a window to the bottom of the window hierarchy. This action can be omitted. This action must have the mnemonic **L**. If this action has an accelerator, it must be **MAlt <F3>** if **<F3>** is available.
Close	**Alt+F4**	Closes a window and removes it from the workspace. This action must have the mnemonic **C**. If this action has an accelerator, it must be **MAlt <F4>** if **<F4>** is available.

A secondary window Menu must have the following selections in the order listed: **Restore**, **Move**, **Size**, and **Close**.

Illustration



Common Bindings

Virtual Key	Default Binding
KWindowMenu	MShift KEscape MAlt KSpace

Related Information

See Chapter 5 for more information on the window Menu and the window manager in general. See Chapter 4 for information about Menu design.

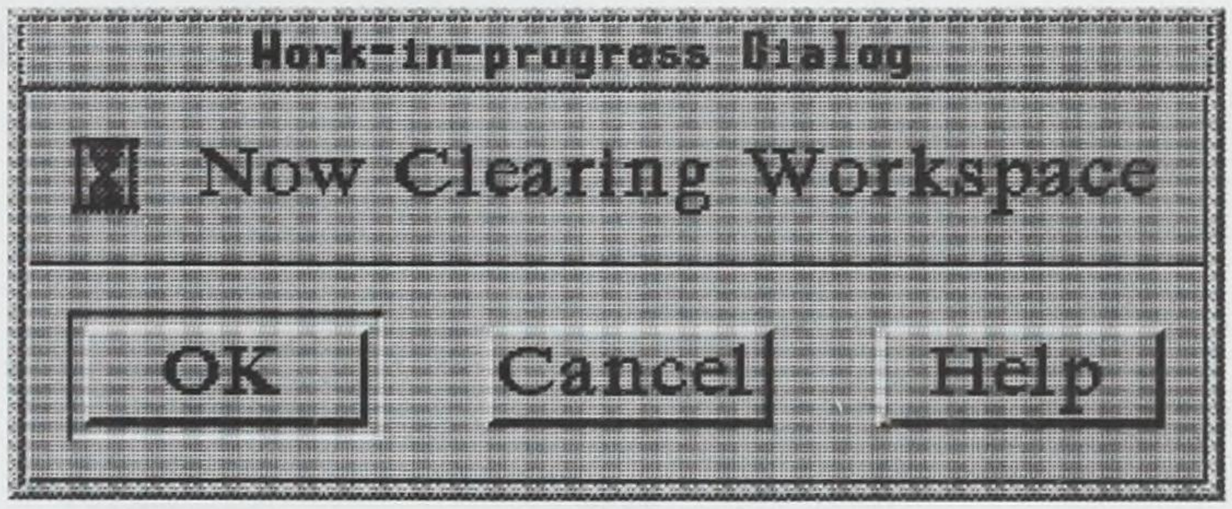
WorkingDialog

Description

A WorkingDialog should be used to show work in progress and give the user an opportunity to cancel the operation. It should not interrupt the user's interaction with the application. It should contain a working symbol, a message, and one of the following button arrangements.

OK Cancel
OK Cancel Help

Illustration



Related Information

For more information, see the reference page for DialogBox.

Appendix A

Common Bindings for Virtual Buttons and Keys

The following tables list the virtual buttons used in this *Style Guide*, along with their common bindings for a mouse with one, two or three buttons.

Table A-1. Virtual Button Bindings for a 3-Button Mouse

Virtual Buttons	Default Bindings
BSelect	<Btn1>
BDrag	<Btn2>
BCustom	<Btn3>
BMenu	BCustom
BExtend	MShift BSelect
BToggle	MCtrl BSelect
BPrimaryPaste	BDrag
BPrimaryCopy	MCtrl BPrimaryPaste
BPrimaryCut	MAlt BPrimaryPaste
BQuickPaste	BDrag
BQuickCopy	MCtrl BQuickPaste
BQuickCut	MAlt BQuickPaste

Table A-2. Virtual Button Bindings for a 2-Button Mouse

Virtual Buttons	Default Bindings
BSelect	<Btn1>
BDrag	1: <Btn2> 2: <Btn2> 3: undefined
BCustom	1: <Btn1> <Btn2> 2: or undefined 3: <Btn2>
BMenu	1: BCustom 2: MCtrl MShift BSelect 3: BCustom
BExtend	MShift BSelect
BToggle	MCtrl BSelect
BPrimaryPaste	BDrag
BPrimaryCopy	MCtrl BPrimaryPaste
BPrimaryCut	MCtrl BPrimaryPaste
BQuickPaste	BDrag
BQuickCopy	MCtrl BQuickPaste
BQuickCut	MAlt BQuickPaste

Table A-3. Virtual Button Bindings for a 1-Button Mouse

Virtual Buttons	Default Bindings
BSelect	<Btn1>
BDrag	undefined
BCustom	undefined
BMenu	MCtrl MShift BSelect
BExtend	MShift BSelect
BToggle	MCtrl BSelect

The following table lists the virtual modifier keys used in this *Style Guide*, along with their common bindings.

Table A-4. Virtual Modifier Bindings

Virtual Key	Default Bindings
MShift	**<Shift>**
MCtrl	**<Ctrl>**
MAlt	**<Alt>**
	<Meta>
	<Extend char>
	<Compose Character>

The following table alphabetically lists the virtual keys used in this *Style Guide*, along with their common bindings. Note that some virtual keys have multiple default bindings, and some have alternate bindings, noted by the word "or." Where a virtual key has multiple default bindings, you should supply all of them. Where a virtual key has an alternate binding it is not necessary that both bindings be available, but you can make both bindings available.

Table A-5. Virtual Key Bindings

Virtual Key	Default Bindings
KActivate	**KEnter**
	MCtrl KEnter
KAddMode	**MShift <F8>**
KBackSpace	**<BackSpace>**
KBackTab	**<BackTab>**
	MShift KTab
KBeginData	**MCtrl KBeginLine**
KBeginLine	**<Begin>** or **<Home>**
KCancel	**<Escape>** or **<Cancel>**
KClear	**<Clear> (optional)**
KCopy	**MCtrl KInsert**
	<Copy>
KCut	**MShift KDelete**

Virtual Key	Default Bindings
	<Cut>
KDelete	<Delete>
KDeselectAll	MCtrl <\>
KDown	<↓>
KEndData	MCtrl KEndLine
KEndLine	<End>
KEnter	<Enter> or <Return>
KEraseEndLine	MCtrl KDelete
KExtend	MShift KSelect
KHelp	<Help> or <F1>
KInsert	<Insert>
KLeft	<←>
KMenu	<Menu> or <F4>
KMenuBar	<F10>
KNextFamilyWindow	MAlt KNextPane
KNextField	KTab
	MCtrl KTab
KNextMenu	MCtrl KDown
	MCtrl KRight
KNextPane	<F6>
KNextPara	MCtrl KDown
KNextWindow	MAlt KTab
KNextWord	MCtrl KRight
KPageDown	<PageDown>
KPageLeft	MCtrl KPageUp
KPageRight	MCtrl KPageDown
KPageUp	<PageUp>
KPaste	MShift KInsert
	<Paste>
KPrevFamilyWindow	MAlt KPrevPane
KPrevField	KBackTab
	MCtrl KBackTab
KPrevMenu	MCtrl KUp
	MCtrl KLeft
KPrevPane	MShift KNextPane
KPrevPara	MCtrl KUp

Virtual Key	Default Bindings
KPrevWindow	MAlt KBackTab
	MAlt MShift KTab
KPrevWord	MCtrl KLeft
KPrimaryCopy	MCtrl KPrimaryPaste
	MAlt KCopy
KPrimaryCut	MAlt KPrimaryPaste
	MAlt KCut
KPrimaryPaste	<vendor defined>
KQuickCopy	MCtrl KQuickPaste
KQuickCut	MAlt KQuickPaste
KQuickExtend	MShift KQuickPaste
KQuickPaste	<vendor defined>
KReselect	MCtrl MShift KSelect
KRestore	MCtrl MShift KInsert
KRight	<→>
KSelect	<Select>
	KSpace
	MCtrl KSpace
KSelectAll	MCtrl </>
KSpace	<space>
KTab	<Tab>
KUndo	MAlt KBackSpace
	<Undo>
KUp	<↑>
KWindowMenu	MShift <Escape>
	MAlt KSelect

Appendix B

OSF/Motif Widgets and Components Correspondence

This appendix shows how actual OSF/Motif widgets correspond to the components described in this *Style Guide*. Gadgets, which are essentially performance-oriented versions of widgets, are not listed in this appendix, but correspond closely to their widget counterparts.

Table B-1. Component to Widget Correspondence

Components	Motif Widgets
Basic Controls:	
Separator	**XmSeparator**
Label	**XmLabel**
PushButton	**XmPushButton** **XmArrowButton** **XmDrawnButton**
ToggleButton	**XmToggleButton** It can also be simulated by **XmPushButton** and **XmDrawnButton**

Components	Motif Widgets
CheckButton	**XmToggleButton** with **XmNindicatorType** set to **XmN_OF_MANY**
RadioButton	**XmToggleButton** with **XmNindicatorType** set to **XmONE_OF_MANY**
CascadeButton	**XmCascadeButton**
OptionButton	An **XmCascadeButton** within an **XmRowColumn** with **XmNrowColumnType** set to **XmMENU_OPTION**
Field Controls:	
ScrollBar	**XmScrollBar**
Scale	**XmScale**
Sash	Private element of **XmPanedWindow**
Text	**XmText** and **XmTextField**
Canvas	**XmDrawingArea** (without children), **XmDrawnButton**
List	**XmList**
Basic Groups:	
Panel	**XmRowColumn** with **XmNrowColumnType** set to **XmWORK_AREA** and composed of basic controls
Menu	**XmRowColumn** with **XmNrowColumnType** set to **XmMENU_PULLDOWN** or a **XmMENU_POPUP**
MenuBar	**XmRowColumn** with **XmNrowColumnType** set to **XmMENU_BAR**
Layout Groups:	
Composite	**XmBulletinBoard**, **XmDrawingArea** (with children), **XmForm**, **XmRowColumn** when **XmNrowColumnType** is set to **XmWORK_AREA** and it is not a basic group
PanedWindow	**XmPanedWindow**

Components	**Motif Widgets**
Framing Groups:	
Frame	**XmFrame**
ScrolledWindow	**XmScrolledWindow**
MainWindow	**XmMainWindow**
DialogBoxes:	
SelectionBox	**XmSelectionBox**
FileSelection Box	**XmFileSelectionBox**
PromptBox	**XmPromptBox**
MessageBox	**XmMessageBox**

Glossary

acceleration — A temporary change in the mouse pointer gain.

accelerator — A key or sequence of keys (typically a modifier key and some other key) that provides a shortcut, immediately accessing a program function.

anchor — A position in a collection of selectable objects that marks one endpoint of an extended selection range.

autorepeat — A means of PushButton activation where a mouse button is pressed and held on a PushButton and the PushButton continues to activate at regular intervals until the mouse button is released.

browse selection — A selection model that allows browsing through single selection collections.

button
A button on a mouse pointing device; mouse buttons can be mapped to the keyboard. A graphical component on a window frame or in a DialogBox that works by pressing it.

button binding
Association of a mouse button operation with a window manager or application function.

cancel
A label given to a PushButton in some DialogBoxes that performs the action of closing the DialogBox without implementing any changes.

cascading Menu
A submenu that provides selections that amplify the parent selection on a PullDown or Popup Menu.

CheckButton
A component used to select settings that are not mutually exclusive. The visual cue to the selection is frequently that the button is filled in or checked.

click
To press and release a mouse button. The term comes from the fact that pressing and releasing most mouse buttons makes a clicking sound.

client
An application program written specifically for the X Window System. Some clients make their own windows; other clients are utility programs.

client area
The area within the borders of a primary window's frame that is controlled by an application.

clipboard
Any device used to store text or graphics during cut-and-paste operations.

close
A label given to a PushButton in some DialogBoxes that performs the action of closing the DialogBox. Close is also used as a selection in Menus to close the window associated with the Menu.

component navigation Moving the keyboard focus among controls within a window.

control panel An area of a window, similar to the control panels in real life, that is used to hold PushButtons and other graphical components.

cursor A graphical image, usually a pipe (|) or block, that shows the location where text will appear on the screen when keys on the keyboard are pressed or where a selection can be made.

default selection An object or action that is specified for selection if no other selection is specified.

desktop Another term for workspace. See **workspace**.

destination cursor The position in a component where paste operations are inserted.

DialogBox A secondary window that the user can display and that contains application components.

dimmed selection A selection that is not currently available.

discontiguous selection A selection model that allows multiple discontiguous selections.

double-click To press and release a mouse button twice in rapid succession.

drag To press and hold a mouse button while moving the mouse.

expert activation A means of activation where double-clicking on a PushButton activates a different action from clicking the PushButton.

explicit focus A keyboard focus model that sends keyboard events to the window or component that was specified explicitly with a mouse button press or a keyboard event.

gain The ratio of the distance the pointer moves to the distance the mouse moves.

graphical user interface A form of communication between users and computers that uses graphics-oriented software such as windows, Menus, and icons to facilitate the interaction.

graphics-like collection A 2-dimentional collection of elements where the location cursor can be either on an element or between two elements.

grayed selection A Menu selection that is not currently available and so has been dimmed.

highlight A graphic technique used to provide a visual cue to the current selection or to the current location of the input focus. Highlighting is frequently accomplished by reversing the video of the selection.

hotspot The area of a graphical image used as a pointer or cursor that is defined as the point of the pointer or cursor.

hourglass A graphical image used to symbolize the passage of time and provide a visual cue that the application is currently performing an operation.

I-beam A graphical image used to represent the location of the mouse pointer in a text entry box and providing a visual cue that text can be entered in an area.

icon A small graphical image used to represent a window. Windows can be turned into icons or minimized to save room or unclutter the workspace.

icon box A window for organizing icons.

inactive A window that does not have the input focus.

insertion cursor The graphical symbol that provides the visual cue to the location of the insertion point.

internal navigation Moving the keyboard focus within a single control.

implicit focus A keyboard focus model that sends keyboard events to the window or component that the mouse pointer is over.

keyboard One of many input devices; the traditional method of entering text into an application.

keyboard focus Indicates the window or component within a window that receives keyboard input. It is sometimes called the input focus.

label The text part of an icon or graphical component.

list box A component that provides users with a scrollable list of options from which to choose.

list-like collection A collection of elements where the location cursor is placed on an element.

location cursor A graphical symbol that marks the current location of the keyboard input focus for selection. Typically, this symbol is a box that surrounds the current object.

lower To move a window to the bottom of the window stack on the workspace.

maximize To enlarge a window to its maximum size.

maximize button A control button placed on an MWM window frame and used to initiate the maximize function.

Menu A list of available selections from which a user chooses.

Menu system A collection of Menus cascading from a single CascadeButton.

MenuBar A rectangular area at the top of the client area of a window that contains the titles of the standard PullDown Menus for that application.

MenuBar system A collection of Menus associated with a MenuBar and the MenuBar itself.

message box The generic name for any DialogBox that provides information, gives the current state of a work in progress, asks a question, issues a warning, or draws attention to an error.

minimize To turn a window into an icon. The term iconify is sometimes used instead of minimize.

minimize button A control button placed on an MWM window frame and used to initiate the minimize function.

mnemonic A single character (frequently the initial character) of a Menu selection, that when the Menu is displayed and the character is pressed on the keyboard, initiates the selection.

modifier key A key that when pressed with another key, changes the meaning of the other key. **<Ctrl>**, **<Alt>**, and **<Shift>** are modifier keys.

mouse A pointing device commonly used in conjunction with a keyboard in point-and-click, object-oriented user interfaces.

mouse button A button on a mouse pointing device. Mouse buttons can be pressed, released, dragged, clicked, and double-clicked.

multiclick To click a mouse button multiple times without moving the pointer.

multiple selection A selection model that allows multiple single selections.

multipress To press a mouse button multiple times without moving the pointer.

multidrag To press a mouse button multiple times without moving the pointer and then drag the mouse pointer.

open To start an action or begin working with a text, data, or graphics file.

paste Inserting data into an area. Pasting is commonly used in reference to text files where a block of text is cut from one area and pasted into another area.

pointer The graphical image that appears on the workspace and represents the current location of a mouse or other pointing device.

pointing device A device such as a mouse, trackball, or graphics tablet that allows users to move a pointer about on the workspace and point to graphical objects.

Popup Menu
A Menu that provides no visual cue to its presence, but simply pops up when users perform a particular action. Popup Menus are associated with a particular area of the workspace, such as the client area of an application, and users must memorize where these areas are.

press
To hold down a mouse button or a key. Note that to hold down a mouse button and move the mouse is called dragging.

previewing
A means of PushButton activation where information about the impending action of a button release is displayed on the button press action.

primary window
A top-level window of an application. Primary windows can be minimized.

PullDown Menu
A Menu that is pulled down from a client application's title bar.

PushButton
A graphic component that simulates a real-life PushButton. People use the pointer and mouse to push the button and start some action.

RadioButton
A graphic component that simulates the buttons on a real-life car radio. Each button represents a mutually exclusive selection. RadioButtons are typically used for setting states or modes.

range selection
A selection model that allows selection of a range of elements.

release
To let up on a mouse button or key that has been pressed. Sometimes it is the press that initiates the action; sometimes it is the release.

resize
To change the height or width of a window.

resize border The MWM window frame part that surrounds the client area of an application and that is used to change the height or width of the window.

restore To return an icon or maximized window to its normal size.

root window See **workspace**.

save To write changes to a data file to a storage device for safekeeping.

screen The physical Cathode Ray Tube (CRT) that displays information from the computer. In most cases in the OSF/Motif environment, screen and workspace are synonymous.

scroll bar A graphical device used to change a user's viewpoint of a list or data file. A scroll bar consists of a slider, scroll area, and scroll arrows. A user changes the view by sliding the slider up or down in the scroll area or by pressing one of the scroll arrows. This causes the view to scroll up or down in the window adjacent to the scroll bar.

secondary window A child window of a primary window.

select To choose an object to be acted upon or an action to be performed.

selection The object or action that is selected. Menus are composed of selection items. Dialog boxes contain components, each of which represents a selection.

Select button The mouse button used to make a selection.

selection cursor See **location cursor**.

single selection A selection model that allows selection of a single element.

slider One of the graphical components of a scroll bar or scale. The slider is the object that is dragged along the scroll area to cause a change.

state A generic term used to describe the condition or mode of an object or action.

submenu A cascading Menu.

system Menu See **window Menu**.

text cursor See **insertion cursor**.

text-like collection A collection of elements where the location cursor is placed between two elements.

title area The area at the top of the window frame immediately beneath the resize border. The title bar has two functions: it contains a title or name that identifies the window and it can be grabbed and dragged to relocate the window.

title bar The bar across the top of an MWM window that consists of the window Menu button, the title area, and the window-control buttons.

transient window A window of short duration such as a DialogBox. The window is displayed for only a short time, usually just long enough to convey some information or get some operational directions.

virtual binding A vendor-independent notation for representing keyboard keys and mouse buttons.

window A data structure that represents all or part of the CRT display screen. Visually, a window is represented as a rectangular subset of the display screen.

window decoration — The frame and window-control buttons that surround windows managed by the a window manager such as the OSF/Motif Window Manager.

window frame — The area surrounding a window. A window frame can consist of a resize border, a window Menu button, a title bar, and window-control buttons.

window manager — A program that controls the size, placement, and operation of windows on the workspace. The window manager includes the functional window frames that surround each window object and may include a separate Menu for the workspace.

window Menu — The Menu that appears when the window Menu button is pressed. The window Menu typically contains selections for restoring, moving, sizing, minimizing, maximizing, and closing the window.

window Menu button — The graphical control button that appears at the left side of the title bar in the window frame.

window navigation — Moving the keyboard focus among windows.

workspace — The CRT screen. The area on which the windows of a user's environment appear. The workspace is sometimes called the desk, desktop, or root window.

Index

A

acceleration, 2-12

accelerators, 3-26, 4-37, 4-39, 7-4, 7-5

actions
    DialogBox, 4-40

activation, 7-5
    basic, 3-24
    component, 3-23
    Menu system, 2-26

active voice, 6-9

active window, 2-2

acts requiring actions, 1-11

add mode, 3-10

addresses, 6-7

affirmative statements, 6-9

alignment
    horizontal, 4-34
    vertical, 4-34

alphanumeric date formats, 6-5

alphanumeric lists, 6-2

anchor, 3-6, 3-8, 3-12

ANS X3.159-1989, 6-1

ANSI C, 6-1

application design, 4-1

application modal, 4-40, 5-5

application title, 5-8

Apply, 4-41, 7-23

arrow pointer, 2-9, 2-11, 7-83, 7-85
    four-directional, 2-10, 7-84

audience, x

autorepeat, 3-29

B

Balance Beam, 3-7

base, 6-1

basic activation, 3-24, 7-5

basic controls, 4-2, 4-10
    Cascade Button, 4-2
    CheckButton, 4-3
    Label, 4-2
    OptionButton, 4-3
    PushButton, 4-2
    RadioButton, 4-3
    Separator, 4-2
    ToggleButton, 4-3

basic groups, 4-2, 4-5, B-1
Menu, 4-5
MenuBar, 4-5
Panel, 4-5

BCustom, 2-6, 2-7, 7-55

BDrag, 2-6, 2-7, 7-55

BMenu, 2-6, 2-7

box
icon, 5-17, 7-50

BQuickCopy, 3-21, 7-94

BQuickCutDrag, 3-21, 7-94

BQuickPaste, 7-94

browse selections, 3-3, 3-5, 3-11, 7-113

BSelect, 2-6, 2-7, 3-24, 7-55

BToggle, 3-7, 3-8

buttons, 5-9
system Menu, 5-11
window Menu, 5-11

C

can, xiii

Cancel, 4-42, 7-24

canceling a selection, 3-13

Canvas, 4-5, 4-9, 4-12, 7-12, 7-33, B-1

CascadeButton, 2-13, 4-2, 7-8, 7-13, B-1

caution pointer, 2-11, 7-85

certification checklist, xiii

CheckBox, 4-10

CheckButton, 4-3, 7-9, 7-15, 7-127, B-1

choosing components, 4-2

Clear, 4-23, 7-29

Click, 2-7

click-to-type, 2-3, 7-43

client areas, 4-13, 5-8

clipboard selection actions, 3-22

Close, 4-19, 5-13, 7-36, 7-131

collating sequences, 6-2

collections
graphics-like, 3-2, 3-3
list-like, 3-2
text-like, 3-2

colors, 1-7

columns
aligning, 4-34

command area, 4-13, 4-15

Command DialogBox, 4-25, 7-20

CommandBox, 7-18

common actions, 7-23

common DialogBox actions, 4-40

Common User Access, ix

compliance conventions, xiii

component activation, 3-1, 3-23

component navigation, 2-21

components
arranging, 4-33

choosing, 4-2
design, 4-52, 7-2
grouping, 4-33
interactive, 4-9
modal, 4-48
multiple-choice, 4-9, 4-10
Scrollable, 2-13
simple, 4-48
single-choice, 4-9, 4-10
text, 7-120
window, 5-6

Composite, B-1

Composition, 4-5, 7-21, 7-58

configurability, 5-2

consistency, 1-8
intraaplication, 1-9

context-sensitive help, 1-11

contrast
screen objects, 1-7

control panel, 4-33

controls, 7-1

Copy, 4-22, 7-28

copying, 3-16, 7-86, 7-87, 7-94, 7-95

corner handles, 5-10

country-specific data formats, 6-3

currency, 6-5

cursor, 2-16
item, 2-18
location, 2-2, 2-3, 2-13, 2-15, 2-16, 2-21, 2-24, 3-10
text insertion, 2-2, 2-17

Cut, 4-21, 7-28

cutting, 7-86, 7-87, 7-94, 7-95

D

data formats
country-specific, 6-3, 6-5
currency, 6-5

date formats, 6-5

decimal separators, 6-4

decoration
windows, 5-6

default action
selecting, 3-27, 7-6, 7-22

default push button, 4-47

defaults, 4-47

delay, 4-51

Delete, 4-23, 7-29

deselecting, 3-14

Design, 1-1

design
applications, 4-1
components, 4-52, 7-2
DialogBox, 4-39
graphical symbols, 6-7
icons, 6-7
pointer shapes, 2-9, 6-7, 7-83
window manager, 5-1

destructive actions, 4-38

dialog box, 4-37

DialogBox groups, 4-6

DialogBox, 2-20, 4-7, 4-8, 4-13, 4-24, 5-2, 7-23, B-1
actions, 4-40
Command, 4-25, 7-20
designing, 4-39

Error, 4-29, 7-31
FileSelection, 4-26, 7-41
groups, 4-2
Information, 4-30, 7-54
location, 4-43
Message, 4-29, 7-71
Prompt, 4-27, 7-88
Question, 4-31, 7-92
Selection, 4-28, 7-117
size, 4-43
Warning, 4-31, 7-129
Working, 4-32, 7-133
direct manipulation, 1-5, 2-6
direction of scanning, xiii
disabling components, 4-48
discontiguous selection, 3-4, 3-8, 3-13, 7-114
double-clicking, 3-28, 7-6
Drag, 2-7
dragging, 3-15, 3-16, 3-17, 3-21
drag-and-drop model, 3-15, 3-17, 7-26

E

edge handles, 5-10
edit menu, 4-16, 4-21, 7-28, 7-65
Edit, 7-65
Enlarge Only, 3-6
Error DialogBox, 4-29, 7-31
ErrorDialog, 7-71
errors, 1-11
Exit, 4-20, 7-36
expert action, 3-28, 7-6
explicit destruction, 1-11
explicit focus, 2-3, 7-43
extension models, 3-6, 3-8

F

feedback, 1-10, 4-45, 4-50
field controls, 2-25, 4-2, 4-4, 4-5, 7-32, B-1
Canvas, 7-33
List, 7-33
Sash, 7-32
Scale, 7-32
ScrollBar, 7-32
text, 7-33
field groups, 2-25
field navigation, 2-24
fields, 2-21
file menu, 4-16, 4-18, 7-35, 7-36, 7-65
File, 7-65
FileSelctionBox, B-1
FileSelection DialogBox, 4-26, 7-41
FileSelectionBox, 7-38
fill, 2-19
fixed layout, 4-33

flexibility, 1-3, 4-52

focus policies, 2-2, 2-3, 2-4, 7-43

focus
    explicit, 2-2, 2-3, 2-4, 7-43
    implicit, 2-2, 2-3, 2-4, 7-43
    keyboard, 2-1, 2-3

formats
    addresses, 6-7
    data, 6-3
    dates, 6-5
    messages, 6-9
    telephone numbers, 6-6
    time, 6-6

four-directional arrow pointer, 2-10, 7-84

Frame, 4-6, 7-45, 7-46, B-1

framing groups, 4-2, 4-6, 7-46, B-1
    Frame, 4-6
    MainWindow, 4-6
    ScrolledWindow, 4-6

G

gain, 2-12

graphics tablet, 2-6, 7-55

graphics, 4-46

graphics-like collections, 3-2, 3-3

group reference, 7-1

grouping
    components, 4-33
    menu selections, 4-38

separators, 6-4

H

handles, 5-10

Help Menu, 7-47

Help, 4-42, 4-51, 7-24, 7-66

help
    activation, 3-27, 7-6
    context-sensitive, 1-11, 4-24
    contex-sensitive, 4-51
    index, 4-24
    menu, 4-17, 4-23, 7-66
    tutorial, 4-24
    version, 4-24

horizontal alignment, 4-34

hot key, 5-2

hotspot, 2-9, 2-11, 7-83

hourglass pointer, 2-10, 7-84

I

icon box, 5-17

icon Menu, 5-16, 7-49

IconBox, 7-50

iconized window, 5-4

icons, 5-4, 5-15, 6-7, 7-52

immediacy of response, 1-5
implicit focus, 2-3, 7-43
Index, 4-24, 7-47
Information Dialog, 7-54
Information DialogBox, 4-30
InformationDialog, 7-71
input devices, 2-1, 2-5, 7-55
input focus, 2-2
input
    keyboard, xi
interactions, 4-45
interactive methods, 4-9
internal window navigation, 2-4
internationalization, 6-1, 6-4, 6-7, 6-9
    collating sequences, 6-2
    data formats, 6-3
    date formats, 6-5
    numbers, 6-6
    telephone numbers, 6-6
intraaplication consistency, 1-8
intraaplication
    consistency, 1-9
item cursor, 2-18
I-beam pointer, 2-9, 7-83

J

joystick, 2-6, 7-55

K

KActivate, 2-26, 3-24
KBackSpace, 3-22
KBegin Data, 2-25
KBeginLine, 2-25
KCancel, 2-28, 3-13
KCopy, 3-22
KCut, 3-22
KDelete, 3-22
KDeSelectAll, 3-14
KDown, 2-22, 2-24, 2-28, 5-19, 7-73
KEnd Data, 2-25
KEnd Line, 2-25
key names
    virtual, xi
keyboard, 2-13
    events, 2-2
    focus, 2-1, 2-2, 2-3, 2-21, 7-43
    input, 2-2, 2-5, 7-55, xi
    navigation, 2-13, 7-73, 7-74, 7-75
    selection actions, 3-15
keyboards, xi, xii
keyboard-based operations, 2-4
keys
    virtual, 7-55, xi, xii
KHelp, 3-27
KLeft, 2-22, 2-24, 2-28, 7-74
KMenu, 2-26, 7-74
KMenuBar, 2-26, 7-74

KNextFamilyWindow, 2-20, 5-14, 7-74
KNextField, 2-21, 5-20, 7-73
KNextMenu, 2-27
KNextWindow, 2-20, 5-14, 7-75
KPaste, 3-22
KPrevFamilyWindow, 2-20, 7-74
KPrevFammilyWindow, 5-14
KPrevField, 2-21, 5-20, 7-73
KPrevMenu, 2-28
KPrevWindow, 2-20, 5-14, 7-75
KPrimaryCopy, 3-19, 7-87
KPrimaryCut, 3-19, 7-87
KPrimaryPaste, 3-19
KQuickCopy, 3-21
KQuickCut, 3-21
KQuickPaste, 3-21, 7-95
KReselect, 7-87
KRight, 2-22, 2-24, 2-28, 5-19, 7-74
KSelect, 2-26
KSelectAll, 3-14
KUndo, 3-22
KUp, 2-22, 2-24, 2-28, 5-19, 7-73

L

Label, 4-2, 4-9, 4-12, 7-8, 7-57, B-1
language direction, xiii
language independent, 6-7
layout groups, 4-2, 4-5, 7-58, B-1
layout, 4-12
    fixed, 4-33
    resizing, 4-34
List, 4-4, 4-10, 7-33, 7-59, B-1
list-like collections, 3-2
localization, 6-1
location cursor, 2-2, 2-3, 2-13, 2-15, 2-16, 2-21, 2-22, 2-24, 3-10
Lower, 5-13, 7-131

M

main component groups, 4-6
main windows, 2-20, 5-2, 5-4
MainWindow, 4-6, 4-8, 4-13, 4-14, 7-46, 7-63, B-1
manipulation, 1-5
    direct, 1-5
maximize button, 5-7, 5-9
Maximize, 5-13, 7-131
Menu system
    activation, 2-26

Menu traversal, 2-25

Menu, 4-5, 4-6, 4-7, 4-8, 4-9, 4-13, 7-10, 7-69, B-1

MenuBar, 2-13, 2-26, 4-5, 4-13, 4-16, 7-10, 7-65, B-1

menus, 5-11, 7-69
  designing, 4-35
  edit, 4-16, 4-21, 7-28, 7-65
  file, 4-16, 4-18, 7-35, 7-65
  help, 4-17, 4-23, 7-47, 7-66
  icon, 5-16, 7-49
  navigation, 2-13
  option, 7-69, 7-76
  options, 4-17
  popup, 7-69
  pull down, 7-69
  selections, 4-38
  structures, 4-37
  system, 7-130, 7-131
  view, 4-17, 7-65, 7-66
  window, 5-3, 5-6, 5-11, 7-130, 7-131

message area, 4-13, 4-15

Message DialogBox, 4-29, 7-71

MessageBox, B-1

messages, 6-9

metaphors
  real world, 1-4

Microsoft Windows, ix

minimize button, 5-7, 5-9

Minimize, 5-13, 7-131

minmized window, 5-4

mnemonics, 3-14, 3-26, 4-36, 4-39, 7-5, 7-72

modeless, 4-40, 5-5

models, 7-1
  input, 2-1
  navigation, 2-1
  selection, 3-1

modes, 4-48

modifiers, xii

motif window manager, 2-2

mouse input, 2-5

mouse pointer shapes, 2-9, 7-83

mouse pointer, 2-3

mouse, 2-2, 2-6, 2-13, 7-55

mouse-based navigation, 2-13

Move, 5-13, 7-130

moving, 3-16

MultiClick, 2-7

MultiDrag, 2-7

multiple cascading submenus, 4-37

multiple MainWindows, 4-14

multiple selection, 3-3, 3-5, 3-12

multiple selections, 7-113

multiple windows, 5-8

multiple-choice components, 4-9, 4-10

MultiPress, 2-7

must, xiii

N

national numbers, 6-6

navigation, 1-6
    components, 2-21
    field, 2-24
    keyboard, 7-73, 7-74, 7-75
    model, 2-1, 2-13
    mouse-based, 2-13
    scrollable components, 2-29, 2-30
    window, 2-4, 5-14

New, 4-18, 7-35

No, 4-41, 7-23

nominal keys, xii

normal mode, 3-10

numbers
    postive and negative, 6-4

numeric date formats, 6-5

O

object-action selection model, 3-1, 3-20

OK, 4-41, 7-23

On Context, 4-24, 7-47

On Help, 4-24, 7-47

On Keys, 4-24, 7-47

On Version, 4-24, 7-47

On Window, 4-24, 7-47

Open, 4-19, 7-35

operation
    keyboard only, 2-5, 7-55
    keyboard-based, 2-4
    mouse only, 2-5, 7-55

OptionButton, 4-3, 4-10, 7-8, 7-76, B-1

Options Menu, 2-13

Options menu, 4-17

Options Menu, 4-35, 7-66, 7-69, 7-76

ordering menu selections, 4-38

ordering
    functions, 1-4
    menu selections, 4-38

outline highlight, 2-17

P

Pack Icons, 5-18

PanedWindow, 4-4, 4-5, 4-35, 7-58, 7-79, 7-99, B-1

Panel, 4-5, 4-11, 7-10, 7-81, B-1

panels
    control, 4-33

Panes, 4-4, 4-35, 7-58, 7-79

parent windows, 5-4

parts
    window, 5-6

Paste, 4-22, 7-29

pasting, 3-16, 7-86, 7-94, 7-95

pointer navigation, 2-13

pointer shapes, 2-9, 6-7, 7-83

pointers, 2-3, 2-6
    arrow, 2-9, 7-83
    caution, 2-11, 7-85
    designing, 2-11
    four-directional, 2-10, 7-84
    hourglass, 2-10, 7-84
    I-beam, 2-9, 7-83
    mouse, 2-3
    question, 2-11, 7-85
    resize, 2-9, 7-84
    sighting, 2-11, 7-85
    track, 2-3
    warping, 2-12
    watch, 2-10, 7-84
    X, 2-9, 7-83

pointing devices, 2-5, 2-6, 7-55

Popup Menus, 2-13, 4-11, 4-35, 7-6, 7-69

positive and negative values, 6-4

POSIX 1003.1, 6-1

posted menu, 2-15

Presentation Manager, ix

Press, 2-7

previewing, 3-29

primary modal, 4-40, 5-5

primary selection, 3-15, 3-17, 3-18, 7-86
    focus-based, 3-18

primary selections, 7-86, 7-87

primary transfer, 3-17, 3-18, 7-86

primary windows, 2-20, 5-2, 5-3, 5-4

PrimaryCopy, 7-86

PrimaryCut, 7-86

PrimaryPaste, 7-86

Print, 4-19, 7-36

progress, 4-51

progressive disclosure, 1-4

Prompt DialogBox, 4-27, 7-88

PromptBox, B-1

proper names, 6-7

prototypes, 1-2

PullDown Menus, 2-13, 4-11, 4-35

PullDown, 7-69

pull-down menu, 4-16, 7-65

PushButton, 4-2, 4-9, 7-8, 7-90, B-1

Q

Question DialogBox, 4-31, 7-92

question pointer, 2-11, 7-85

QuestionDialog, 7-71

quick selection, 3-20, 7-94

quick transfer, 3-15, 3-16, 3-20, 7-94

QuickCopy, 7-95

QuickCut, 7-95

QuickExtend, 7-95

R

RadioBox, 4-10

RadioButton, 4-3, 7-9, 7-96, 7-127, B-1

range selection, 3-4, 3-6, 3-12, 7-113

reading direction, 6-8

real world metaphors, 1-4

real-estate driven, 2-3

reference pages, 7-1

related items, 4-38

Release, 2-7

ReSelect, 3-6

Reset, 4-41, 7-24

resize border, 5-7, 5-10

resize pointer, 2-9, 7-84

resizing layout, 4-34

response immediacy, 1-5

restore, 5-9

Restore, 5-12, 7-130

Retry, 4-41, 7-24

S

Sash, 4-4, 7-32, 7-79, 7-99, B-1

Save As, 4-19, 7-35

Save, 4-19, 7-35

Scale, 4-4, 7-32, 7-103, B-1

scanning direction, 6-8, xiii

scroll bar, 4-15

scrollable component navigation, 2-13, 2-29, 2-30

ScrollBar, 4-4, 7-32, 7-107, B-1

ScrollBars, 4-13, 4-15

ScrolledWindow, 4-6, 7-46, 7-111, B-1

scrolling, 2-29, 2-30

secondary selection mechanism, 3-20

secondary windows, 2-20, 5-2, 5-3, 5-4
- application modal, 5-5
- interaction, 5-5
- modeless, 5-5
- modes, 5-5
- primary modal, 5-5
- system modal, 5-5

SelectAll, 3-14

selection actions, 3-1
- clipboard, 3-22

Selection DialogBox, 4-28, 7-117

selection model, 3-1, 3-3, 3-4
- object-action, 3-20

selection, 4-38
- browse, 3-3, 3-5, 3-11, 7-113
- canceling, 3-13
- default, 7-22
- deselecting, 3-14
- discontiguous, 3-4, 3-8, 3-13, 7-114
- keyboard, 3-10
- mnemonics, 3-14

model, 7-113
multiple, 3-3, 3-5, 3-12, 7-113
primary, 3-17, 3-18, 7-86, 7-87
quick, 3-20
range, 3-4, 3-6, 3-12, 7-113
secondary, 3-20
single, 3-3, 3-5, 3-11, 7-113
undoing, 3-13
SelectionBox, 7-115, B-1
selections
quick, 7-94
Separator, 4-2, 7-8, 7-119, B-1
separators
decimal, 6-4
grouping, 6-4
thousands, 6-3
sequences
collating, 6-2
shapes
pointer, 2-9, 7-83
should, xiii
sighting pointer, 2-11, 7-85
single selection, 3-3, 3-5, 3-11, 7-113
single-choice components, 4-9, 4-10
Size, 5-13, 7-130
spring-loaded, 2-13
Stop, 4-41, 7-24
structures
menu, 4-37
submenus
multiple cascading, 4-37
system Menu button, 5-11
system menus, 7-130, 7-131
system modal, 4-40, 5-5

T

tab groups, 2-21
Tear-Off Menus, 4-7, 5-6
telephone numbers, 6-6
Text, 4-5, 4-9, 4-12, 7-33, B-1
component, 7-120
text
cursor, 2-17
insertion cursor, 2-2
translation, 6-9
text-like collections, 3-2
thousands separators, 6-3
time formats, 6-6
title area, 5-7, 5-8
title bar, 5-8
toggle models, 3-8
ToggleButton, 4-3, 7-8, 7-15, 7-96, 7-127, B-1
track ball, 2-6, 7-55
track listener, 2-3
track pointer, 2-3
transient windows, 2-20, 5-2, 5-3
translating text, 6-9
traversal
keyboard, 2-5

Tutorial, 4-24
types of windows, 5-2
typographic conventions, xi

U

undo, 1-11, 3-13
Undo, 4-21, 7-28

V

values
    poitive and negative, 6-4
vertical alignment, 4-34
vertical panes, 4-35
view menu, 4-17, 7-65, 7-66
View, 7-65
virtual bindings, xi
virtual buttons, A-1, xii
virtual key mechanism, 7-55
virtual keys, 7-55, A-3, xi, xii
virtual modifier keys, A-3
visual cue, 2-2

W

Warning DialogBox, 4-31, 7-129
warnings, 1-11, 4-51
warping, 2-12
watch pointer, 2-10, 7-84
window control buttons, 5-10
window family, 2-20
window manager
    design, 5-1
    functions, 5-6, 5-10
window Menu button, 5-7, 5-11
window menus, 7-130, 7-131
window navigation, 2-20
window panes, 7-58, 7-79, 7-99
windows
    active, 2-2
    components, 5-6
    decoration, 5-6
    frame components, 5-9
    iconized, 5-4
    main, 2-20, 5-2, 5-4, 7-63
    menu bar, 4-16
    menu, 5-3
    minmized, 5-4
    modes, 5-5
    multiple, 5-8
    navigation of, 7-75
    navigation, 2-4, 5-14, 7-74, 7-75
    panes, 4-35
    parent, 5-4
    primary, 2-20, 5-2, 5-3, 5-4, 7-63
    resize border, 5-10

secondary, 2-20, 5-2, 5-3, 5-4
transient, 2-20, 5-2, 5-3
types, 5-2

Working DialogBox, 4-32, 7-133

WorkingDialog, 7-71

X

X pointer, 2-9, 7-83

Y

Yes, 4-41, 7-23

Z

zoom feature, 2-12

Notes

Notes

Notes

Notes

OPEN SOFTWARE FOUNDATION

INFORMATION REQUEST FORM

Please send me the following:

( ) OSF Membership Information

( ) OSF/Motif™ License Materials

( ) OSF/Motif™ Training Information

Contact Name _____

Company Name _____

Street Address _____

Mail Stop _____

City _____ State _____ Zip _____

Phone _____ FAX _____

Electronic Mail _____

MAIL TO:

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

Attn: OSF/Motif™

For more information about OSF/Motif™, call 617-621-8755.

OSF/Motif™

Release 1.1

Style Guide

TITLES IN THE OSF/Motif SERIES:

OSF/Motif Programmer's Guide

OSF/Motif Programmer's Reference

OSF/Motif Style Guide

OSF/Motif User's Guide

Application Environment Specification (AES)
User Environment Volume

Printed in U.S.A.

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

Prentice-Hall, Inc.

ISBN 0-13-640616-5